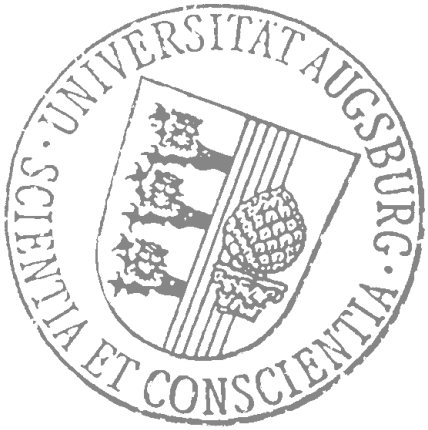
Mooop – A Generic Integration of Object-Oriented and Ontological Models



Christoph Frenzel

TR 2010-14

(basierend auf der Masterarbeit von Christoph Frenzel)

Dezember 2010

Universität Augsburg
Institut für Informatik

Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
http://www.informatik.uni-augsburg.de/

Abstract

Object-oriented programming (OOP) is a widely adopted technology for implementing information systems because it provides various means for modelling structure and behaviour. Nevertheless, some domains like Health Care and Life Science are so large and complex that OOP is not suited for their modelling. In comparison, the Web Ontology Language (OWL) provides various expressive modelling constructs and is used to formulate large, well-established ontologies in these domains. OWL can not, however, express or describe behaviour and, thus, can not be used alone to build applications. Therefore, an integration of both paradigms, which leverages the advantages of each, is desirable, yet not easy to accomplish: differences in their semantics induce an impedance mismatch that needs to be taken into account.

This work presents Mooop (Merging OWL and Object-Oriented Programming), an approach for the generic integration of OWL ontologies into OOP-based models. It introduces hybrid objects, which represent both an OOP and OWL model entity, in order to integrate both paradigms. More precisely, it provides an adoptable mapping between the OWL model and the OOP model. In this way, it creates a coherent hybrid model which can be easily exploited by the application. We have developed a prototype of Mooop to show its advantages in two case studies.

Contents

Chapter 1

Introduction

Computerised information systems are a pillar of our modern information society. They store, retrieve, and process information of huge amounts and high complexity which materialises itself in complicated domain models. They are characterised by complex static structures, sophisticated dynamic behaviour and numerous constraints on the stored information.

Over the years, several programming paradigms and languages have been introduced to support developers of ever bigger and more complex systems. Thereby, *Object-oriented programming* (OOP) is a famous and widely adopted technology for modelling information systems, especially business applications. It provides various means for structuring the problem domain and, so, is well suited for sophisticated domain models. However, its main advantage lies in its expressiveness of the dynamic behaviour of applications. The popularity of this approach is underpinned by numerous *object-oriented programming languages* (OOPL) like Java, C++, or Ruby and lots of standard frameworks for developing *object-oriented* (OO) applications like Java EE[1], ASP.NET[2], or Lift[3].

Nevertheless, some domains, like Health Care and Life Science, are so large and complex that OOP is not suited for their modelling. For instance, anatomical models are characterised by a vast number of concepts with complex logical constraints which are hard to express in OOP. In comparison, the *Web Ontology Language* (OWL) provides various expressive modelling constructs which suit the requirements of these domains very well. Additionally, its foundation in *Description Logics* (DL) provides OWL with reasoning abilities, i.e., the inference of additional implicit knowledge from explicit knowledge. Unsurprisingly, domain experts have already used OWL to formulate large, well-established models called *ontologies* like GALEN [61] or SNOMED CT [50]. On the contrary to its great expressive power concerning the static structures of domain models, OWL does not allow the modelling of dynamic behaviour. Therefore, it alone can not be used to build applications.

This comparison reveals that an integration of both paradigms, which lever-

[1] `http://www.oracle.com/technetwork/java/javaee/overview/index.html`, accessed: 20 Oct 2010

[2] `http://www.asp.net`, accessed: 20 Oct 2010

[3] `http://liftweb.net`, accessed: 20 Oct 2010

ages the advantages of each, is desirable. The goal is to preserve the key features of OOP and OWL in order to benefit from both. However, this is not easy to accomplish because differences in their semantics induce an impedance mismatch that needs to be taken into account.

This mismatch is in general hard to overcome, however, some OOPLs provide means to support the bridging of the gap between both paradigms: especially dynamic languages with elaborated reflection mechanisms and prototype-based languages are suited for the integration of OWL. They allow an object to dynamically change its type and structure at runtime and, therefore, enable the direct representation of OWL entities in an OO model. However, a successful integration approach also has to be easily adoptable by many OOP developers. Therefore, it is desirable to support a mainstream OOPL like Java, C#, or C++. In order to provide a concrete implementation of the presented integration approach, we concentrate on the widely adopted OOPL Java. However, this decision has some implications on the final solution: on the one hand, the previously envisioned integration is not feasible since Java is missing some essential features, e.g., the modification of object types at runtime. On the other hand, it provides the developer with mature tools like *integrated development environments* (IDE) and debuggers which allow a very efficient development.

This work presents our approach for the generic integration of OWL ontologies into OOP-based model: *Merging OWL and Object-Oriented Programming* (Mooop). We introduce *hybrid objects* which are entities represented in both the OWL ontology and the OOP model. Additionally, Mooop provides a flexibly customizable mechanism for linking the hybrid objects into the OWL ontology and the OOP model. In this way, Mooop creates a coherent hybrid model which can be easily exploited by the application and preserves the features of OWL and OOP as needed. Another aim of Mooop is the separation of the definition of an integrated model and the logic which bridges the gap between OWL and Java. The former is supposed to be performed by application developers through the implementation of *hybrid classes* which define hybrid objects. These hybrid classes and their attributes are mapped to concepts in the ontology via Java annotations. Listing 1.1 shows an example of the idea for an integration. However, the bridge between OWL and Java, which defines the semantics of a hybrid class, is supposed to be developed by specialised ontology developers.

```
1  @OwlClass("OwlPizza")
   public class Pizza {
3    @OwlType
     private String[] types;
5
     @OwlProperty
7    private Map<String, Object> indirectProperties;

9    @OwlProperty(name="hasName")
     private String name;
11 }
```

Listing 1.1: The vision of hybrid classes in Mooop.

This work focusses on the design of the Mooop concept and evaluates it. Although it is a very interesting and important factor, the development of a

methodology for building integrated models using Mooop is not in the scope of this work. So, after a discussion of the general concepts for integrating an OWL ontology into an OOPL model we analyse the requirements for a generic integration approach and, subsequently, introduce our solution concept: Mooop. In order to provide a decent evaluation, a prototype has been developed which design is outlined.

Chapter 2

Domain Modelling

This chapter describes the type of models which should be integrated, and introduces and compares the two modelling formalisms uses by Mooop: Java and OWL.

2.1 Domain Models

Computer science and especially software engineering can be seen as the science of building software models [10]. The most famous definition of a model in this context is the one by Rothenberg:

> "Modeling in its broadest sense is the cost-effective use of something in place of something else for some purpose. It allows us to use something that is simpler, safer, or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality."[62]

The most important characteristic of a model is that it is an *abstract representation* of reality. On the one hand, abstract means that it concentrates on only some interesting aspect of reality. Thereby, different models can represent the same system at different levels of abstraction, i.e., with a different degree of omission of facts about reality. On the other hand, a model is a representation because the model is not reality, but instead the elements which build up the model are related to objects from the real world. The goal is that a model fulfils the contextual substitutability, i.e., "a model should be able to answer a given set of questions in the same way the system would answer these same questions."[10]

In this work, we concentrate on a special kind of models: *domain models*. A domain model is a software model which concentrates on the problem domain a program should work on. Hence, the model does not contain any implementation

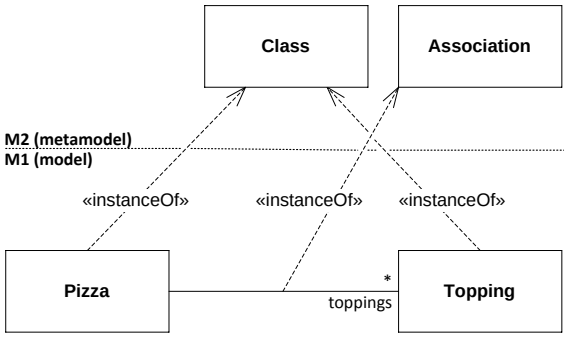


Figure 2.1: Relation of model and metamodel (see [45, p. 17]).

or platform specific details about the software. Thus, it is similar to the *platform independent model* [43, p. 2-6] defined in the *Model Driven Architecture* (MDA) standard by the *Object Management Group* (OMG).

Furthermore, the domain models we are interested in are *bipartite* domain models. They model the problem domain from two points of view: a static viewpoint and a dynamic viewpoint [4]. This distinction is very common and also used in the famous OMG *Unified Modelling Language* (UML) [46, p. 15]. The static part models the *structure* of the domain, i.e., it defines valid states of the real world which can occur and separate them from invalid, impossible states. These states usually comprise objects with their values and relations between them. The dynamic part models the *behaviour* of the domain, i.e., it defines valid transitions from an input state to an output state. These transitions usually describe how the objects, values, and relations of the input state change to reach an output state. The transitions are often labelled in order to express that the behaviour is triggered by a special event in the real world.

A model is expressed in some modelling language. This language defines a syntax for expressing models and a semantics of the syntactical elements. A modelling language again can be seen as a model for all the models expressed in it. This leads to the notion of *metamodel* as defined by Seidewitz:

> "... a metamodel makes statements about what can be expressed in the valid models of a certain modeling language."[64]

Figure 2.1 depicts a small UML example which shows that a metamodel (syntactically and semantically) defines the elements for expressing models which, vice versa, conforms to the metamodel. In the four-layer metamodel hierarchy of the OMG, *M1* refers to the model and *M2* to the metamodel [45, p. 16]. However, there is no modelling language which provides means to express every aspect of the world. Hence, most modelling languages concentrate on specific aspects and, therefore, are suited for specific types of models. Famous representatives of modelling languages are the OMG UML for software systems, the Entity-Relationship Diagram metamodel for databases, OWL for knowledge systems, and Java for object-oriented programs.

2.2 Java Modelling

Java is commonly known as an OOPL especially suited for implementing information systems. However, a Java program, i.e., a collection of classes and interfaces, can also be seen as a model which conforms to the Java modelling language. But in order to make the model executable a Java program usually combines both a domain model and an implementation model which contains, e.g., the user interface or a persistence framework. In this work, we focus on the domain model part of a Java model. Furthermore, the term OO and related terms refer to Java if not explicitly said otherwise.

Java is a strongly typed, class-based OOPL [20, p. 1] which allows the definition of OO models. The basic elements of OO models are *objects* which exchange *messages* [60, p. 10 et seq.]. Objects have an explicit, fixed, and unique *object identity* (OID) and are characterized by a state, i.e., a collection of variable values, and behaviour, i.e., a collection of actions performed after a message reception. This characteristic feature is often referred to as *encapsulation* because an object combines data with behaviour which works on this data. The behaviour is accessible through *methods* which all together define the *interface* of an object, i.e., a collection of messages the object understands. In a class-based OOPL an object is created by instantiating a *class*. It defines the structure of the object, i.e., the object's variables and methods, and, thus, can be seen as a template for objects. Hence, an object is always the instance of a class, or, in other words, the object has a type. A class can inherit the structure from another class and, thus, become a *subclass*. This allows the subclass to extend or overwrite the behaviour of the superclass. Although very common, class-based languages are not the only kind of OOPLs: Another approach are prototype-based OOPLs which allow the direct manipulation of the structure of objects and object cloning. A strongly typed OOPL is characterized by the fact that the type of an object is defined at compile-time. Additionally, it is forbidden to send a message to an object which it does not understand. Therefore, a strongly typed OOPL can prevent a lot of errors caused by not understood messages.

The expressiveness of Java concerning the structure of a domain is limited. It allows the definition of base classes and subclasses which are extending the definition of another class. However, Java only allows single inheritance, i.e., a subclass can extend at most one other class. This restriction is relaxed for interfaces: a class can inherit the method signatures of several interfaces. Properties are defined within a class along with the type of the property values. Java only distinguishes between single properties and unlimited multiple properties. Java objects have to be explicitly typed, i.e., upon instantiation the type of an object must be explicitly known. Furthermore, an object can be the instance of only one class and this type can not change. Java implicitly assumes a *unique name assumption* (UNA) concerning the OID: two objects with different OID are different and two different objects with the same OID an not exist.

On the contrary to the structure, Java has a great expressive power concerning the behaviour. The transitions are implemented in the methods of Java classes and triggered by events defined as the method signatures of Java interfaces[1]. The methods are implemented in an imperative, Turing complete

[1] Notice that a Java class defines a class and an interface with the same name and methods.

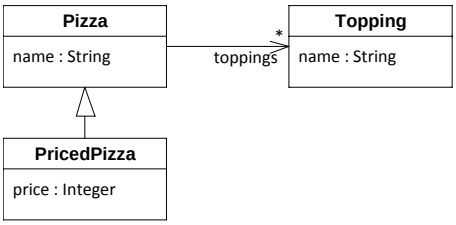


Figure 2.2: Example pizza model for illustrating the different Java models.

language with enormous arithmetic and logical capabilities. As an OOPL, Java offers a dynamic binding based on a single dispatch which enables a basic polymorphism. It is interesting to notice that the behavioural expressiveness of Java is so extensive that the limited structural expressiveness is enhanced by runtime constraints expressed in methods [36].

The former discussion shows that the Java modelling language is based on the *closed world assumption* (CWA) [48]. The CWA assumes complete information about the real world. Therefore, it follows that every fact that is not true must be false. For instance, if an object o *is not defined* to be an instance of class A then o *is not* an instance of A.

The Java modelling language is very flexible and allows the definition of several kinds of models of a domain. According to Puleston et al. there are [59]:

- Direct models
- Indirect models
- Hybrid models

The following descriptions of these models are based on [59] and use the little example depicted in Figure 2.2: there are pizzas (class Pizza) and toppings (class Topping). Both have a property name of type String. A pizza is associated with several toppings. A PricedPizza is a special pizza which has an additional property price of type Integer.

2.2.1 Direct Models

Direct models are also called *traditional object-models* [75] because they conform to the traditional way of object-oriented modelling: Java classes represent concepts in the real world and Java objects represent specific individuals of these concepts in the real world. The classes define the shape, i.e., the properties and methods of their instances whereby the objects define the state, i.e., the values of properties of a specific instance.

A direct model for the little pizza example is quite simple: it is the Java equivalent of the example model in Figure 2.2, i.e., the UML classes become Java classes and UML properties become Java attributes. At runtime, the system consists of objects of these classes. Figure 2.3 depicts the objects and their relations for a pizza Margherita.

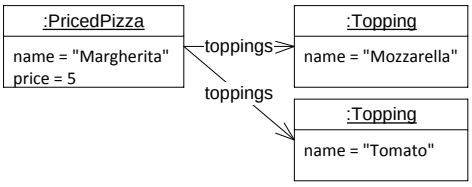


Figure 2.3: Object model of a pizza Margherita in a direct model.

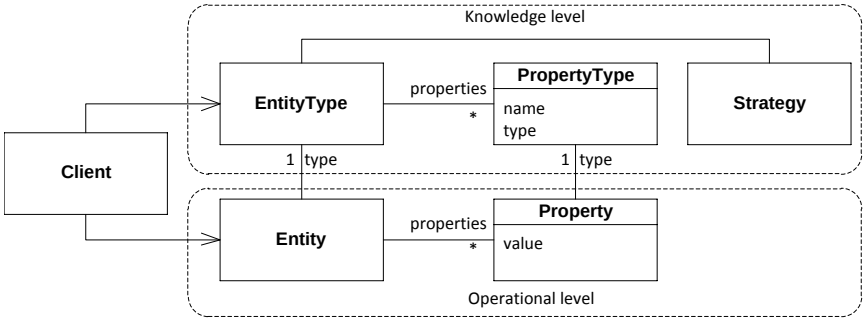


Figure 2.4: Class model of a generic indirect model.

Direct models are modelling the world based on Java classes. The main advantage of this is that they provide a domain-specific *application programming interface* (API) for accessing and manipulating the model to the developer. The behaviour is encapsulated into the Java classes. This enables an easy development of domain-aware software in a way which OO-developers are used to. Furthermore, the domain-specific API allows the definition of type constraints enabling a type-safe development. The main disadvantage of direct models is that they are static. They represent one specific domain model and if the domain alters, the direct model has to be altered as well. For a Java program, this means that the Java source code has to be adapted and recompiled.

2.2.2 Indirect Models

Indirect models are also called *adaptive object-models* (AOM) [75]. As shown in Figure 2.4, an AOM splits the Java model into a *knowledge level* often referred to as metadata, and an *operational level* [17]. The Java classes in an AOM are representing *meta-concepts* in the real world. Hence, the Java class model can be seen as a metamodel. The model itself is represented by the objects of the Java classes. Thereby, objects of the class EntityType represent concepts in the real world and, thus, are similar to Java classes in a traditional object model. Objects of the class Entity represent specific individuals and, thus, are similar to objects in a traditional object model. Notice that each object of Entity has an association to an EntityType which determines the type of the instance. An EntityType can also define its properties whereas an Entity defines the values for the properties of its type. The behaviour of an EntityType is defined by its associated Strategies.

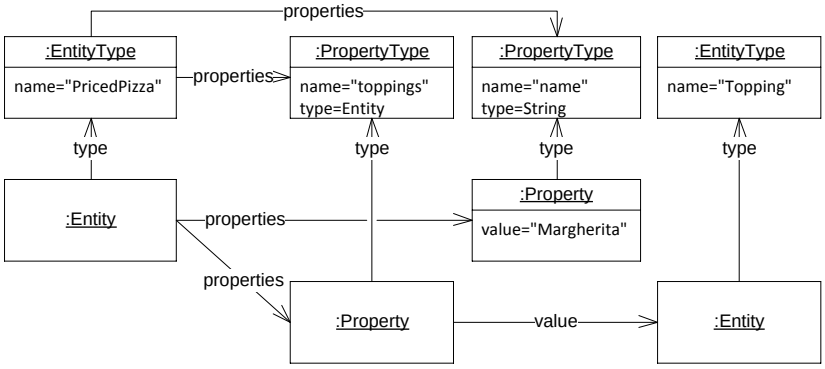


Figure 2.5: Object model of a pizza Margherita in an indirect model.

Indirect models are more complicated than equivalent direct models. This becomes obvious when implementing the little pizza example. The class model is the one already shown in Figure 2.4. The classes have to be instantiated in order to build both the knowledge level objects and operational level objects. The indirect character of the model leads to numerous objects and relations between them. Figure 2.5 shows only a part of the resulting object model for a pizza Margherita.

Indirect models are modelling the world based on objects. Their main advantage is that they are very flexible and can change their structure at runtime. For instance, new concepts can be created by instantiating EntityType and the properties of a concept can be changed by modifying its associations with PropertyType objects. Therefore, indirect models can be easily adapted to changes of the domain. Furthermore, the domain model does not have to be modelled in Java but can be loaded at runtime from, e.g., a database. The main disadvantage of indirect models is their domain-neutral API. It obscures the structure of the model and makes the access very complicated. Furthermore, the behaviour is not encapsulated and has to work on a generic Entity. This is similar to structured, pre-OO languages which separated data and behaviour. Finally, the type-safety of Java is lost. For instance, it is not possible to restrict the type of a method parameter to a specific model class, i.e., to a specific object of Entity.

2.2.3 Hybrid Models

A hybrid model is a combination of direct and indirect model. The basic idea is that a small number of *top level concepts* is modelled directly. These are the core and most important concepts of the domain. However, the vast number of specializations of these core concepts is modelled indirectly. Hence, there are Java classes for both concepts and meta-concepts. The goal of a hybrid model is to create "a model that can exploit the strengths of each side to compensate for weaknesses of the other, or to accommodate different skill sets and preferences of the modellers."[59]

Figure 2.6 shows a hybrid model for the little pizza example. Pizza and Topping have been identified as top level concepts and, thus, are represented

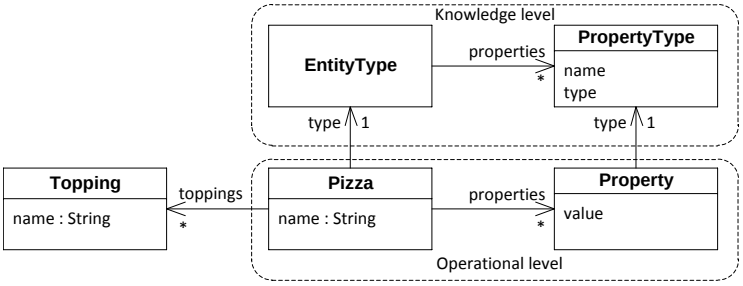


Figure 2.6: Class model of a hybrid model for the pizza example.

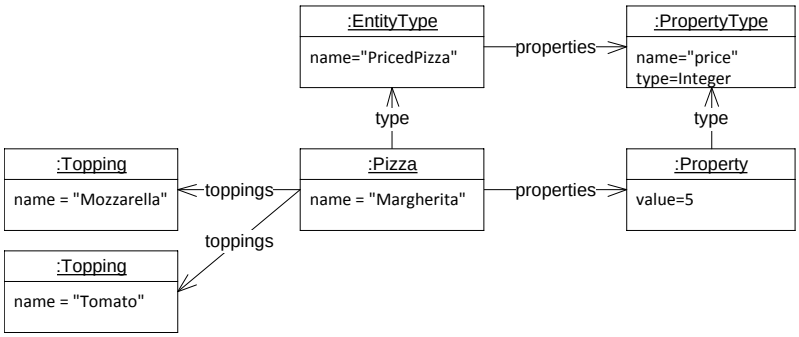


Figure 2.7: Object model of a pizza Margherita in a hybrid model.

in a direct manner. This becomes obvious in comparison to the direct model depicted in Figure 2.2. However, the Pizza class can be specialised whereby the specialisations (in this case PricedPizza) are represented in an indirect manner. Therefore, besides its direct properties, the Pizza class also has indirect properties and types. This can be seen in comparison to the indirect model depicted in Figure 2.4. The class Pizza is called a *hybrid class* because it combines the direct and indirect modelling approach. The instances of hybrid classes are referred to as *hybrid objects*.

The hybrid character of the approach can be seen at runtime (see Figure 2.7): the Java type (Pizza) and the directly represented attributes (name, toppings) of the hybrid object are fixed and provide a domain-specific type system. However, the indirectly represented specialised class (PricedPizza) and properties (price) can change during runtime.

Hybrid models are modelling the world based on classes and objects. This approach has a lot of advantages: on the one hand, the directly modelled top level concepts provide a domain-specific API and type-safety for most use cases. The behaviour can be encapsulated in these core Java classes. On the other hand, the indirectly modelled parts of the domain model can be adapted to changes of the domain at runtime. However, this comes at the price of an increased complexity of the modelling because its hard to determine which parts of the domain model should be modelled directly and which parts indirectly. Even worse, the advantages of the hybrid model depend heavily on the correct division.

2.3 OWL Modelling

The OWL Web Ontology Language (OWL) is a language for capturing knowledge in form of an *ontology*. Studer et al. defines,

> "An ontology is a formal, explicit specification of a shared conceptualisation."[69]

This means that an ontology is a machine readable (*formal*) model of some aspect of the world (*conceptualisation*) which contains *explicitly* defined concepts and relations between them and is accepted by a group of users (*shared*). OWL is based on DLs which are defined by Baader and Nutt as,

> "... a family of knowledge representation (KR) formalisms that represent the knowledge of an application domain (the 'world') by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description)."[5, p. 47]

DLs are subsets of *first-order logic* (FOL) which makes them less expressive than FOL. DLs are based on the *open-world assumption* (OWA) [5, p. 72]: they assume that the information about the real world is incomplete. Therefore, it follows that every fact that is not explicitly modelled to be true or false is unknown. For instance, if an individual i *is not asserted* to be neither an instance of class A nor an instance of class $\neg\mathsf{A}$ then the relation of i to A is *unknown*. Consequently, both interpretations $\mathsf{o} \in \mathsf{A}$ and $\mathsf{o} \notin \mathsf{A}$ are valid.

The OWL 2 Web Ontology Language (OWL 2) is the current standard by the *World Wide Web Consortium* (W3C) and the successor of the original OWL Web Ontology Language. It has been developed to provide an extended expressiveness [21]. OWL 2 is based on the DL $\mathcal{SROIQ}(\mathbf{D})$ [30] which offers a well defined and decidable semantics of the modelling elements, and sound and complete decision procedures. The OWL 2 standard defines several syntaxes like a *functional syntax*, a *OWL/XML syntax*, and the human readable *Manchester Syntax* [24]. There is even a specification by the OMG for a graphical syntax called *Ontology Definition Metamodel* (ODM) [44]. In this work we use the Description Logics syntax introduced in [28] which is shown in Appendix A. Furthermore, in this work we are solely using OWL 2 ontologies and, therefore, we are using OWL to refer to the OWL 2 Web Ontology Language.

From a technological point of view, OWL is based on the *Resource Description Framework* (RDF) in the *Semantic Web Architecture* [27]. RDF represents data in form of triples containing a subject, predicate, and object [41]. Thereby, the elements of the triple are identified using *Uniform Resource Identifiers* (URI). In this way, RDF can be used to represent complex knowledge. For instance, Listing 2.1 shows an RDF example which expresses that the website `http://www.example.org/index.html` was created by the employee with the number 85740 on August 16, 1999, and is written in English. However, RDF is lacking an exact description what the different elements mean, e.g. that English is a language. However, this can be provided by an OWL ontology.

```
1 <http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/
      creator> <http://www.example.org/staffid/85740> .
  <http://www.example.org/index.html> <http://www.example.org/terms/
      creation-date> "August 16, 1999" .
3 <http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/
      language> "en" .
```

Listing 2.1: RDF example showing 3 data triples.

The expressiveness of OWL to model structural features of a domain is extensive [71]. Similar to Java, OWL distinguishes between a concept level called *TBox* and an instance level called *ABox* [67]. The former defines the OWL classes and OWL properties of an *OWL model* (that is an ontology) and the latter the OWL individuals. OWL allows the definition of *atomic classes* like Pizza. A class can be defined to be a *subclass* of another class like PricedPizza $\sqsubseteq$ Pizza. Similarly, a class can also be *equivalent* to another class like PricedPizza = PizzaWithPrice. A *property* is a first class citizen of an OWL model and, therefore, defined stand-alone. OWL distinguishes between object properties, which are a relation between two OWL individuals, and data properties which are a relation between an OWL individual and an OWL literal, i.e., a data value like an integer. Properties can be organized in a subclass hierarchy as well. Furthermore, OWL allows the restriction of the range and the domain of a property to special classes. For instance, a customer relation can be defined as $\geqslant 1\mathsf{hasTopping} \sqsubseteq \mathsf{Pizza}$; $\top \sqsubseteq \forall \mathsf{hasTopping}.\mathsf{Topping}$ meaning that hasTopping is a relation between a Pizza and a Topping. However, these restrictions should not be interpreted as constraints: if an OWL individual i is not asserted to be an instance of Topping then an assertion $\langle i_1, i\rangle \in \mathsf{hasTopping}$ would not lead to an error but to the implication that $i \in \mathsf{Topping}$. Besides these basic modelling elements, OWL also allows the definition of *complex classes* which are combinations of atomic classes and property restrictions. They are the main reason for the great expressiveness. Property restrictions induce restrictions on the property assertions of individuals of an OWL class. For instance, $\mathsf{ToppedThing} \sqsubseteq \exists \mathsf{hasTopping}.\top$ defines that a ToppedThing has a relation hasTopping to some other individual. Complex classes are build by combining complex classes, atomic classes, or property restriction using the logical operators and ($\sqcap$), or ($\sqcup$), and not ($\neg$). This allows definitions like $\mathsf{TomatoPizza} \sqsubseteq \mathsf{Pizza} \sqcap \exists \mathsf{hasTopping}.\mathsf{Tomato}$. Notice that these complex classes also allow multiple inheritance. This can be extended to a *general concept inclusion* (GCI) which can be seen as a general restriction on the model. For instance, $\mathsf{Pizza} \sqcap \exists \mathsf{hasTopping}.\mathsf{Tomato} \sqsubseteq \mathsf{VegatarianFood} \sqcup \mathsf{NonVegatarianFood}$ defines that the individuals which are both Pizza and $\exists$hasTopping.Tomato are also either VegatarianFood or NonVegatarianFood.

The OWL individuals are defined in the ABox. An OWL individual can have several types which are the atomic or complex classes it belongs to. However, the types of an OWL individual do not all have to be explicitly defined. On the contrary, OWL allows for an implicit typing. For instance, $\langle i, i_1\rangle \in \mathsf{hasTopping}$ implies $i \in \mathsf{Pizza}$. Furthermore, property assertions for object and data properties define that the OWL individual is in a property relation with another individual or data value. Because of the OWA, it is also possible and often necessary to define *negative property assertions*, i.e., property values an individual

does not have. OWL does not support a UNA and, therefore, it is also possible to define equal and disjoint individuals for an OWL individual.

On the contrary to the great expressiveness of structures, OWL can not express any behaviour at all [71].

Its logical foundation provides OWL with a very important capability: *reasoning*, i.e., the inference of implicit knowledge from explicit knowledge. Reasoning is performed by *reasoners* like FaCT++ [72], Pellet [66], or HermiT [65] which provide their functionality through several *reasoning services*. Examples for these services are [1]:

Consistency checking verifies that the complete ontology is consistent, i.e., it contains no contradictory facts.

Concept satisfiability verifies that a class is satisfiable, i.e., the class can have instances.

Classification computes the complete subclass hierarchy.

Realization computes all types of an OWL individual.

In order to distinguish between explicit knowledge, so called *asserted* knowledge, and implicit knowledge, so called *inferred* knowledge, we write $\alpha \in \mathcal{O}$ to denote asserted knowledge, and $\mathcal{O} \models \alpha$ to denote inferred knowledge whereby α is some OWL axiom and $\mathcal{O}$ is an OWL ontology. Notice that an ontology can be seen as a set of axioms.

Reasoning services enable an important feature of OWL often referred to as *post-coordination* [51, p. 91]. Usually, an ontology consists of numerous atomic classes which define the concepts of the world. This is called pre-coordination. For instance, imagine the definition of a nut allergy as $\mathsf{NutAllergy} = \mathsf{Allergy} \sqcap \exists \mathsf{causedBy}.\mathsf{Nut}$. The idea of post-coordination is that not all concepts of the real world are defined in the ontology as atomic classes. Instead, the user of the system defines the concepts at runtime as anonymous complex classes using two or more atomic concepts from the ontology [74]. Thus, a concept like almond allergy is not defined in the ontology, however, it can be represented by the complex class $\mathsf{Allergy} \sqcap \exists \mathsf{causedBy}.\mathsf{Almond}$. Since almond is a nut, this complex class is classified as a $\mathsf{NutAllergy}$ which can imply further knowledge.

In order to use post-coordination, the possibilities for defining complex classes at runtime should be restricted. This is called *sanctioning* [8]. Sanctioning restricts the classes, properties, and property values for the definition of a complex class. Imagine that we extend the above reasoning example by the general definition of an allergy as $\mathsf{Allergy} \sqsubseteq \exists \mathsf{causedBy}.\mathsf{Allergen}$. If a user wants to define the almond allergy from above, he or she starts by defining the class $\mathsf{Allergy}$. Subsequently, the user should be guided to define the $\mathsf{causedBy}$ property because this is a *reasonable* refinement of the class. Accordingly, if the user starts to define a special nut allergy by using the class $\mathsf{NutAllergy}$ then the system should guide him to define the $\mathsf{causedBy}$ property with a value which is a Nut. Notice that this is just an example of a sanctioning and that concrete sanctioning rules are domain specific. Sanctioning restricts the definition of not reasonable concepts in an ontology and, thereby, introduces some kind of constraint and CWA-based behaviour.

Java model element	OWL model element
Class	Class
Inheritance hierarchy	Subclass hierarchy
Attribute	Property
Objects	Individual
Data type	Literal

Table 2.1: Related modelling elements of Java and OWL.

2.4 Object–Ontological Impedance Mismatch

In order to enable a combined modelling of a domain in Java and OWL, it is important to compare the expressiveness of both approaches. It is obvious that both modelling languages are not congruent since Java allows the modelling of behaviour which OWL does not. However, a detailed analysis of their structural modelling means is still interesting.

Superficially seen, both approaches share a lot of similar modelling elements and "most of the differences between ontologies and object models are just in naming."[71] This is shown in Table 2.1 and also summarized by Knublauch et al. as,

> "Domain models consist of classes, properties and instances (individuals). Classes can be arranged in a subclass hierarchy with inheritance. Properties can take objects or primitive values(literals) as values."[36]

However, although these modelling elements are similar, their semantics is actually different and not combinable. This mismatch results from the different and incompatible foundations of Java and OWL [48]: on the one hand, Java is based on the CWA and constraints, and, on the other hand, OWL is based on OWA and DL. The induced mismatch is comparable to the mismatch between object-oriented and relational data models and, therefore, we call it *object-ontological impedance mismatch*.

Oren et al. point out that the object-ontological impedance mismatch is the subsumption of five mismatches shown in Table 2.2 [48][49]. A Java object is the instance of exactly one class, but an OWL individual can be the member of several classes. Java classes can inherit structure and behaviour from at most one superclass. Java interfaces do only provide method signatures but do not define any structure and, hence, they are neglected here. On the contrary, an OWL class can not only be the subclass of several other classes but can also be defined as a complex class which is not expressible in Java. A Java object strictly conforms to its type, i.e., it has exactly the structure and behaviour defined by the type. Hence, a Java object has exactly the properties and restrictions defined in the type, not less or more. On the other hand, an OWL individual does not have to conform to its types. Thus, the asserted structure of the OWL individual can be different from the defined structure of the types of the OWL individual as long as the structure definitions do not contradict each other. The properties of a Java object are contained as attributes in a Java class. They are encapsulated in the class and can only be used in the context of the class.

Feature	Java	OWL
Class membership	Single	Multiple
Class inheritance	Single	Complex classes
Object conformance	Strict	Lax
Property	Encapsulated in class	Self-contained
Runtime evolution	Static classes	Dynamic TBox

Table 2.2: Mismatches between Java and OWL models according to [48] and [49]

However, OWL properties are first class citizens of the model and self-contained. Thus, they can be used in any context and, so, an OWL individual can define a value for a property which is not contained in one of its types. The last feature in Table 2.2 refers to the common usage patterns of Java models and OWL models: the class part of Java models is often seen as static because changes in the classes demand a recompilation of the Java program. Hence, the only dynamic part of the Java model at runtime are the objects. On the contrary, an OWL model does not have this distinction: a change of the TBox requires no more effort than a change of the ABox. Therefore, the TBox is regularly changed and adopted to the domain.

The different semantics of the Java modelling language and the OWL modelling language also induce differences in modelling patterns. OWL models often contain numerous classes which encode fine-grained states in which the individuals can be at runtime. On the contrary, Java models often contain fever classes encoding basic concepts. Fine-grained runtime states are defined as queries, i.e., behaviour which can be executed at runtime. As an example imagine a pizza which is complete if it contains at least one topping. In an OWL model there would be a specific class for a complete pizza which encodes this semantics. However, in Java there would be only the class for a pizza with a method which checks whether it is complete or not.

Chapter 3

OWL–Java Integration Approaches

There are already several approaches for integrating ontological and object-oriented models. In general, the integration can be performed in two directions:

- Integrate an object-oriented model into an ontological model by extending an ontological modelling language with some language for expressing behaviour, e.g., the *Semantic Web Rule Language* (SWRL) [29].
- Integrate an ontological model into an object-oriented model by representing ontological concepts in the object-oriented language. Therefore, usually some kind of restriction concerning the ontological or object-oriented expressiveness has to be imposed.

This work concentrates on the latter integration direction. More precisely, we concentrate on the integration of an OWL ontology into a Java model.

Happel and Seedorf propose a categorisation of different integration concepts which distinguishes between the use of ontologies at development time and runtime [22]. The former is called *ontology-driven development* (ODD). Approaches of this category transform an OWL model into a Java model using sophisticated and complex translations which should preserve as much knowledge as possible. In this case, OWL is seen as a modelling language like UML and only used at development time. At runtime the OWL model is not accessed and, hence, these approaches provide neither runtime access to the OWL model nor reasoning. Representatives of this category are the *Ontology Creator* [32], *OntoJava* [15], and the *ontology compiler*[1] [19]. The latter category comprises *ontology-based architecture* (OBA) approaches which use the ontology at runtime and, therefore, provide runtime OWL model access and reasoning. Since this work concentrates on OBA, we neglect ODD approaches in the following.

The main challenge for the numerous approaches for integrating OWL into Java is overcoming the object-ontological impedance mismatch. Thereby, differ-

[1]Notice that the ontology compiler creates a Visual Basic or C# model. However, both languages are similar to Java concerning their modelling capabilities and, thus, the resulting model could be expressed in Java, as well.

ent types of OO models can be employed to represent the ontological entities. Puleston et al. propose a categorisation of the different integration approaches depending on this feature [59]. Hence, the domain models of approaches from different categories are distinguished in whether the Java classes are representing OWL entities or OWL meta-entities. They distinguish three categories:

Direct Integration uses a direct model so that Java classes represent OWL model entities.

Indirect Integration uses an indirect model so that Java classes represent OWL metamodel entities.

Hybrid Integration uses a hybrid model so that Java classes represent OWL model and OWL metamodel entities.

Since we are focussing on approaches for integrating OWL models into Java models, concepts like *ActiveRDF* [48], *SWCLOS* [38], or *Zhi#* [52] are not considered because they make use of special features of other programming languages. ActiveRDF uses the dynamic scripting languages Ruby. This decision has been felt after performing an evaluation which showed that dynamic scripting languages with strong meta-programming facilities are especially well suited for overcoming the object-ontological impedance mismatch. ActiveRDF dynamically creates a domain specific API for OWL entities at runtime. This is possible because Ruby is not strongly typed and, so, allows to send arbitrary messages to objects which are intercepted by the ActiveRDF object manager and translated into an RDF query over the ontological model. This feature makes indirect accessible types and properties almost needless. SWCLOS is a "Semantic Web Processor developed on top of CLOS"[38], the Common Lisp Object System, which can be seen as a dynamic OOPL. The processor can be used to translate an OWL ontology into a CLOS model which supports a lot of OWL features including reasoning [37]. Zhi# is a compiler framework on top of the OOPL C# and allows the integration of external type systems like *XML Schema* (XSD) or OWL into the host language. The result is a seamless integration of internal and external types which allows reasoning. However, Zhi# extends the host language with further constructs and, hence, requires a special compiler. Another approach by Clark and McCabe uses the multi-paradigm language *Go!*, which integrates logic, functional, object-oriented, and imperative programming, in order to represent an OWL ontology [12]. However, this can be seen as another ODD approach.

In the rest of this chapter we present the categories by Puleston et al. in more detail and show their advantages and disadvantages. In order to exemplify the differences, we use an example ontology shown in Figure 3.1. This ontology is an OWL representation of the example in Chapter 2.2 (see Figure 2.2). Notice that, following OWL conventions, the properties have been prefixed with *has*.

3.1 Direct Integration

A direct integration approach represents information from the OWL model in a direct model in the OOPL. Hence, Java classes represent OWL classes (TBox)

$$\begin{aligned} \mathsf{Pizza} &\sqsubseteq \exists \mathsf{hasName}.\mathtt{String} \sqcap \exists \mathsf{hasTopping}.\mathsf{Topping} \\ \mathsf{Topping} &\sqsubseteq \exists \mathsf{hasName}.\mathtt{String} \\ \mathsf{PricedPizza} &\sqsubseteq \mathsf{Pizza} \sqcap \exists \mathsf{hasPrice}.\mathtt{Integer} \end{aligned}$$

Figure 3.1: OWL ontology representing the pizza example model from Figure 2.2.

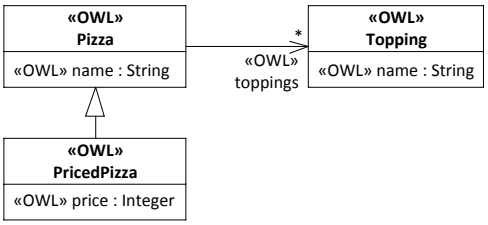


Figure 3.2: Java model for the directly integrated example ontology.

and Java objects represent OWL individuals (ABox). A sanctioning mechanism has to be employed in order to derive the attributes of a Java class which represents an OWL class. In other words, the sanctioning mediates between the lax object conformance of OWL and the strict object conformance of Java. The main goal of a direct integration approach is to preserve the features of the OOPL, e.g., strict object conformance and type safety.

In order to integrate the small pizza ontology, the Java model has to contain Java classes for interesting OWL classes. Therefore, the classes Pizza, Topping, and PricedPizza which is a subclass of Pizza would be created. A reasonable sanctioning would create Java attributes for all existential restrictions of OWL classes. Hence, Pizza and Topping have a name, and PricedPizza has a price. The resulting Java model is depicted in Figure 3.2. Thereby, Java classes and Java attributes which are directly integrated are stereotyped with *OWL*. This convention will be used throughout the rest of this work. The integration framework has to take care that changes in the Java model are reflected in the ontology. For instance, upon instantiating a Pizza in Java an OWL individual with the type Pizza has to be created in the ontology, and upon setting the value of a Java attribute a corresponding OWL property assertion has to be added to the ontology. Notice that, besides the mapped attributes, Java classes can also contain pure Java attributes.

The main advantage of the direct integration approach is its use of the direct model. Hence, the direct integration benefits from a domain-specific API, encapsulated behaviour, and type safety. This allows the easy development of domain-specific software. However, the approach has also severe disadvantages. First, it is often necessary to create numerous Java classes and interfaces because, usually, OWL models contain far more classes than normal Java models would. This can lead to confusion of the developers. Second, the reasoning capabilities are limited since the model is static: it is not possible to change the type or structure of an object at runtime. Thus, if the type of an OWL individ-

ual changes through reasoning, the developer has to instantiate a new Java class for it. Consequently, the reasoning is usually only used during instantiation of an OWL individual but not during manipulation. Third, it allows no decoupled evolution of the OWL model and the Java model. If the OWL model changes then the Java model usually has to be adapted and the software recompiled. In summary, the direct integration can be seen as a *static integration*.

The most notable representatives of this category are RDF-mapping approaches like *Jenabean* [13], *So(m)mer* [68], and Elmo [39]. They can be used to selectively integrate OWL concepts into a Java model by annotating Java classes with OWL classes and their Java attributes with OWL properties. This is similar to the *Java Persistence API* (JPA) approach which deals with relational databases. At runtime, objects of the annotated Java classes represent individuals from a RDF model and provide an API to manipulate them. Older approaches like *RDFReactor* [73] and *Jastor* [70] do not allow to select the integrated ontological classes, but instead translate the complete ontology into a Java model. However, at runtime the Java objects are also representing individuals from a RDF model.

Elmo is an especially interesting approach because it offers means for a dynamic OWL-based method dispatch: it allows a method polymorphism based on the information in the RDF model. A developer can write behaviour classes which implement a domain-specific interface. These classes provide implementations for the methods of an interface and are annotated with an OWL class. Upon a call of a method of the interface for an entity, the framework collects all behaviour classes which match the types of the entity and executes the methods in a chain of responsibility [18]. Although this is an interesting feature, the execution of the methods is not intuitive and can lead to unexpected behaviour.

An representative which does not use Java annotations is agogo [54]. It allows the definition of Java classes and their mapping to OWL concepts via an own *domain specific language* (DSL). This definition is translated into Java source code which can be used to access the OWL model through a domain-specific API. However, the most interesting feature is that it allows the definition of complex SPARQL [57] queries in a RDF model, which can be bound to methods of Java classes. This allows to exploit ontology design patterns, i.e., generic problem solutions comparable to the OO design patterns [18], and, at the same time, hide them from the Java developers.

3.2 Indirect Integration

An indirect integration approach represents information from the OWL model in an indirect model in the OOPL. Hence, the metaclasses of OWL are represented as Java classes and the M1 entities of the OWL model (TBox and ABox) are represented as Java objects at runtime. In general, no sanctioning mechanism is necessary because the interrelated Java objects can represent a lax object conformance. The main goal of an indirect integration approach is to preserve the features of OWL, e.g., lax object conformance and reasoning abilities.

The Java model for integrating the example pizza ontology depends on the framework used for integrating the OWL model. Figure 3.3 depicts a simple

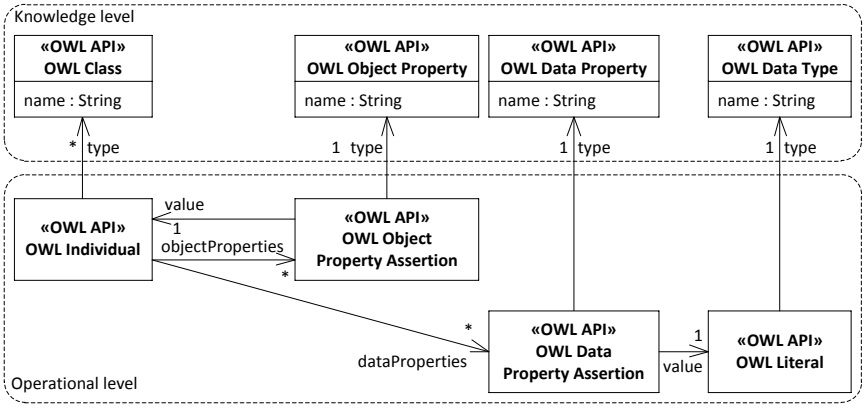


Figure 3.3: Data model of a framework for the indirect integration of OWL into Java.

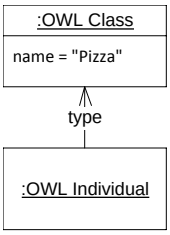


Figure 3.4: Object model of an OWL individual of type Pizza using an indirect integration approach.

OWL data model which is aligned with the syntactical concepts of OWL: there are OWL Individuals which can have several OWL Class objects as their types and several OWL Object Property Assertion and OWL Data Property Assertion objects, respectively, as their properties. Notice that this model solely represents the asserted facts of the ontology. For most indirect OWL integration frameworks a special reasoner component has to be used to gather inferred knowledge about an OWL entity. This model can be seen as a specialized version of the indirect model shown in Figure 2.4. Notice that all classes of the integration framework are stereotyped with *OWL API*. This convention will be used throughout the rest of this work. As with all indirect models, the domain model of the pizza example can not be seen in the class diagram but solely as objects of the framework classes at runtime. Figure 3.4 depicts a newly instantiated OWL Individual which is given the type Pizza by assigning it to the OWL Class with the value Pizza for the attribute name. Notice that the OWL Individual has no properties because their are no asserted properties, yet.

This approach is well suited for ontology-centric systems because it allows the most flexible access to the OWL model. Therefore, it is used by low level ontology access APIs like the *OWL API* [25] or *Jena* [14]. They provide a generic, domain-neutral API for accessing ontological knowledge.

Since an indirect model can change at runtime, they allow the usage of

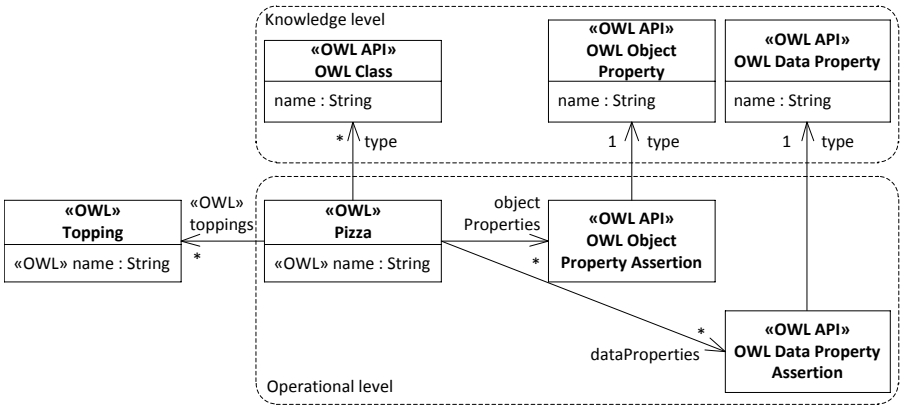


Figure 3.5: Java model for the hybridly integrated example ontology.

sophisticated reasoning services which dynamically change the entities of the OWL model by, e.g., inferring additional type and property assertions for an OWL individual. Furthermore, the loading of the OWL model at runtime also enables a decoupled evolution of the OWL and Java model. In summary, the indirect integration approach can be seen as a *dynamic integration* which facilitates the development of domain-neutral programs. However, there are also shortcomings of this approach, which can be traced back to the disadvantages of indirect models: complicated model access, non-encapsulated behaviour, and no type safety.

3.3 Hybrid Integration

A hybrid integration approach represents information from the OWL model in a hybrid model in the OOPL. Hence, Java classes are representing both metaclasses and classes of the OWL ontology (TBox), and Java objects are representing classes and individuals from the OWL model (TBox and ABox). Since the hybrid integration combines direct and indirect integration, a sanctioning mechanism has to be employed as for the direct integration. The main goal of a hybrid integration approach is to preserve as many features as possible of both the OOPL and OWL.

The Java model for a hybrid integration of the example pizza ontology into Java is a combination of the direct and indirect integration. Figure 3.5 depicts the class model. Pizza and Topping are directly integrated OWL classes with directly integrated properties as Java attributes. Furthermore, Pizza is a hybrid class because it also has indirect properties and types. The OWL class PricedPizza is indirectly integrated and, hence, is represented as an instance of the class OWL Class at runtime.

In the same way the hybrid model combines the advantages of the direct and indirect model, the hybrid integration combines the advantages of the direct and indirect integration: domain-specific API, type safety, encapsulated behaviour, runtime manipulation of the model allowing sophisticated reasoning and decou-

pled model evolution. However, the hybrid integration also inherits the main disadvantage of the hybrid model: increased complexity of the integration.

The *Core Model-Builder* [58] is, to the best of our knowledge, the most sophisticated framework for a hybrid integration of an OWL model into Java. It is actually conceived as a generic hybrid integration framework for several kinds of modelling formalisms into Java. However, the OWL integration is the most prominent integration and offers a lot of features like configurable hiding of OWL concepts, dynamic constraints on indirect integrated attributes, and a complex sanctioning for deriving the structure of an OWL class. Thereby, the sanctioning derives the attributes including types and constraints for an OWL class from the OWL model. The directly accessible attributes are bound to the attributes of the directly integrated Java model class and the rest is indirectly accessible. Another interesting feature is that the Core Model-Builder also provides a domain neutral access to the model, namely the network version.

Another interesting hybrid integration approach is *TwoUse* (Transforming and Weaving Ontologies and UML in Software Engineering) [53]. It is an MDA-based framework which allows the creation of a *hybrid model*, a combination of UML and ODM which contains pure Java model classes, pure OWL model classes and combined Java-OWL classes. A number of transformations translate the hybrid model into an OWL model file and Java source code. The runtime integration of the two models is performed by So(m)mer and, thus, is in principle direct. However, TwoUse allows to query specialized, indirectly integrated types of an entity. These types can be the result of a classification of the entity and its properties in the OWL model. But these specialized types can not be modified and the structure of an entity is fixed and can not be dynamically extended with further attributes. Therefore, this approach is very limited.

Chapter 4

Requirements Analysis for Mooop

This chapter presents an analysis of the requirements for a generic integration of ontological models into OO models by breaking the abstract vision down into distinct features which can be validated. This is accomplished by defining and analysing different categories of OWL-OO applications, and deriving concrete requirements for the Mooop approach from the results.

4.1 OWL–OO Application Categories

The usage patterns of ontologies in applications are vast. However, the wide range of OO applications using ontologies can be structured into three categories:

Ontology editors are applications which concentrate on the OWL domain model. They are characterized by an extensive and complex ontology which can be queried and manipulated in a generic fashion. Therefore, the OO part of the application only serves as an executable viewer. Representatives of this category are generic OWL editors like Protégé [3] or Swoop [33], but also OWL model-centric applications like the TAMBIS system [6] which aims to provide a transparent access to disparate biological databases.

RDF data-based applications concentrate on the OO domain model. They are characterized by both a complex behaviour and an extensive OO domain model which data should be stored and shared on the semantic web. Thus, the OO model reflects a part of the ontological model. A RDF file is used as a persistence store, comparable to a relational data base, and the reasoning services are exploited for integrating different OO domain models. Representatives of this category are mainly semantic web applications like the Travel Agency example by Knublauch [35], the FOAFMap [56] which uses the FOAF ontology [11], or the UK region checker [47].

Ontology-based applications concentrate on both the ontological and the OO domain model and integrate them. They are characterized by a rich and evolving ontology and a complex, concept-specific behaviour which is expressed in the OO domain model. Reasoning is conducted regularly in order to derive new knowledge form the model. Representatives of this category are the Patient Chronicle Model [58] which is introduced later in this chapter, and the context-aware application framework presented in [67].

It is obvious that the different application categories are suited for different integration approaches (see Chapter 3): on the one hand, an ontology editor is a software system typically using a generic indirect integration. The direct integration, on the other hand, is best suited for RDF data-based applications. However, because of its flexibility, a hybrid approach can be used to implement all application types, tough, taking its complexity into account, it suits ontology-based applications the most.

Furthermore, the three application types differ in their degree of integration of the OWL and OO model. The loosest integration can be found in ontology editors and the tightest in ontology-based applications. Hence, ontology-based applications require the most sophisticated integration approach and, so, are the most interesting use cases for Mooop. Therefore, in the following, we concentrate on ontology-based applications and their specific requirements.

Ontology-based applications are most complex application category and require a sophisticated hybrid integration. On the one hand, it is apparent that the utilisation of an indirect integration approach leads to an awkward and inconvenient model access through the generic OO model classes. Furthermore, the extensive behaviour can not be encapsulated along with the data in objects but has to be implemented in a purely structured fashion. On the other hand, a direct integration approach is not flexible enough to keep pace with a constantly evolving ontology because the OO model has to constantly adapt as well. Furthermore, these approaches do not offer sophisticated reasoning mechanism for deriving new knowledge about an entity during processing.

Current hybrid integration concepts (see Chapter 3.3) are not suited for a broad variety of ontology-based applications. The TwoUse approach is limited in its features since it solely provides ontological type queries as an indirect means of integration. It allows no manipulation of the types of OWL individuals and no indirectly integrated properties. Hence, all dynamic ontological information has to be encoded as OWL classes which limits the functionality. As a result, TwoUse has in general the same limitations concerning the implementation of ontology based applications as direct integration approaches.

The Core Model Builder offers means for a comprehensible hybrid integration including querying and manipulation of type and property assertions. The integration of the OWL model into the Java model utilizes a sophisticated sanctioning mechanism which is based on OWL classes but neglects OWL individuals. Therefore, it is an elaborated framework for applications like the Patient Chronicle Model software. However, the adaptation to other integration styles is quite complex and the mapping of OO entities to OWL entities is mixed with functional code which can lead to confusion.

4.2 Case Studies

In the following we present two case studies for ontology-based applications. They should make things more concrete. Furthermore, they were used to identify requirements, show the feasibility of the Mooop concept and evaluate the prototypical implementation of Mooop.

4.2.1 Pizza Configurator

The first running example is a *pizza configurator* which allows the creation of a custom pizza and dynamically calculates its price. It is inspired by *The Manchester Pizza Finder*[1] developed by Matthew Horridge at the University of Manchester. The pizza finder is based on the OWL API and the *Pizza Ontology*[2]. It can be used to get suggestions for pizzas from the ontology which satisfy some user preferences defined by included and excluded pizza toppings, and the size of the pizza. Appendix B shows two screenshots depicting this workflow. Behind the scenes, the pizza finder creates a class expression based on the desired pizza toppings by building a conjunction over expressions of the form $\exists \mathsf{hasTopping}.\mathsf{T}$ for an included Topping T, expressions of the form $\neg\exists \mathsf{hasTopping}.\mathsf{T}$ for an excluded Topping T, and an expression of the form $\mathsf{hasPizzaSize} : \mathsf{S}$ for the size class of the pizza. This expression is classified and the subclasses extracted which yield the result. The Manchester Pizza Finder can be seen as an ontology editor application because the ontology provides the central model and the application does not have any sophisticated behaviour except for the creation of the pizza query of the ontology.

The pizza configurator extends this idea in order to be a ontology-based application. It offers the possibility to configure a pizza and calculate its price. Thereby, the price can be either the sum of the prices of the selected toppings or, if it is a predefined pizza, the price of the pizza. The logic is that the price is generally the sum of the prices of the toppings but can be overwritten by a price of a defined pizza. This logic, however, can not be expressed with the means of OWL but has to be implemented in the OOPL. In order to support this scenario, the aforementioned Pizza Ontology has to be adopted to the new requirements. First, a functional OWL data property hasPrice with the range integer is defined. Second, in this example all toppings are given the price 1 by defining that the superclass of all Toppings, PizzaTopping, is a subclass of $\mathsf{hasPrice} : 1$. Third, certain pizzas are defined by an equivalent class axiom defining exactly the toppings and a subclass axiom defining the price. Thus, a pizza P with $n \in \mathbb{N}$ toppings $\mathsf{T}_1 \ldots \mathsf{T}_n$ and the price p would be defined through axioms shown in Figure 4.1 which build up a GCI. The resulting ontology can be found as a part of the pizza configurator case study (see Appendix D).

A typical workflow for the pizza configurator is simple: first, the user adds and removes several toppings to an initially empty pizza. Thereby, the system has to ensure that it is not possible to add one topping more than once. When the user has finished this step, the pizza is classified an the price for the pizza

[1] `http://www.co-ode.org/downloads/pizzafinder`, accessed: 1 September 2010

[2] `http://www.co-ode.org/ontologies/pizza/pizza-latest.owl`, accessed: 1 September 2010

$$
\begin{aligned}
\mathsf{P} \equiv & \mathsf{Pizza} \\
& \sqcap \exists \mathsf{hasTopping}.\mathsf{T}_1 \sqcap \cdots \sqcap \exists \mathsf{hasTopping}.\mathsf{T}_n \\
& \sqcap \forall \mathsf{hasTopping}.(\mathsf{T}_1 \sqcup \cdots \sqcup \mathsf{T}_n) \\
\mathsf{P} \sqsubseteq & \mathsf{hasPrice} : p
\end{aligned}
$$

Figure 4.1: Axioms for defining a pizza in the pizza configurator ontology.

is calculated. Thereby, the system reasons whether the configured pizza is a defined one with a defined price or whether the price has to be calculated using the topping prices. Furthermore, additional information like the spiciness of the pizza are inferred.

4.2.2 Medical Patient Model

The second running example is a *medical patient model* which is inspired by the *Patient Chronicle Model* and its implementation based on the *Core Model-Builder* (see Chapter 3.3) as shown in [58]. The Patient Chronicle Model derives patient histories from health records to facilitate sophisticated analyses on the data. Thus, time is an important factor throughout the application. However, this feature can be neglected for a study of the OWL-OO integration since it is solely modelled and implemented in the OOP language. Therefore, the medical patient model concentrates on a *Clinical Problem Glimpse*, a record of a single examination of a patient at one point in time. Figure 4.2 shows a simplified Clinical Problem Glimpse which is a hybrid class. It can be seen that is has pure Java attributes like time point, directly accessible OWL properties like location, but also indirectly accessible OWL properties like stage and sub-stage (not visible in the class diagram). The clinical problem glimpse supports complex interactions and sanctioning. Every change of a value of a property triggers a reasoning which can result in changes of properties values, or the addition or removal of whole properties to or from the set of indirectly accessible properties. The information about these manipulations are derived from the ontology. An example workflow, which is derived from [2], could be as follows:

1. Set the type of Clinical Problem Glimpse to *cancer*. This results in the addition of a several new indirect properties like *stage*.
2. Set the locus of Clinical Problem Glimpse to *breast*. Thus, it can be inferred that the problem is a *breast cancer*.
3. Set the stage of Clinical Problem Glimpse to *stage III*. This results in the addition of a new indirect property *sub-stage*.
4. Set the type of Clinical Problem Glimpse to *leukemia*. This results in the removal of the property *sub-stage*. Furthermore, it can be inferred that the locus of the problem is the *hematopoietic system*.

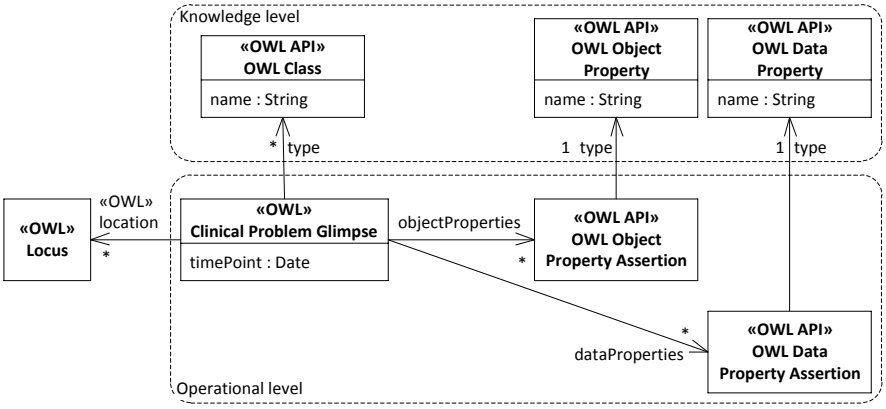


Figure 4.2: Simplified Java model of a Clinical Problem Glimpse.

4.3 Requirements

Mooop is supposed to be a generic approach for the integration of OWL ontologies into Java models. Ontology-based applications are the most interesting type of OWL-Java software for Mooop because they demand the most sophisticated integration. The analysis of ontology-based applications and the shortcomings of current approaches for their implementation reveals important functional requirements for the Mooop framework. They can be classified as follows:

1. Hybrid integration
   (a) Preserve important features of Java
   (b) Preserve important features of OWL
2. Flexible integration
3. Declarative integration

A hybrid integration is a powerful approach for integrating an OWL model into a Java model suitable for a wide range of applications. Therefore, Requirement 1 demands this promising integration approach for Mooop. A general concept should integrated both the ABox and the TBox of the ontology, thus, enabling the exploitation of OWL classes, OWL properties, and OWL individuals. The point of intersection between OO and OWL should be *hybrid classes* which are representing atomic or complex OWL classes in the Java model. The instances of these Java classes, the *hybrid objects*, should be linked to a specific OWL individual. Thus, the hybrid object consists of a combined state, consisting of a pure OO state from itself and a pure OWL state from the linked OWL individual. It should be possible to easily access the whole hybrid state from Java. Although the integration of the TBox is easier to accomplish, it seems important for a generic approach to also integrate the ABox. The reason is not solely that the usually contains information which can be directly related to Java objects, but it also offers a higher degree of expressiveness and more

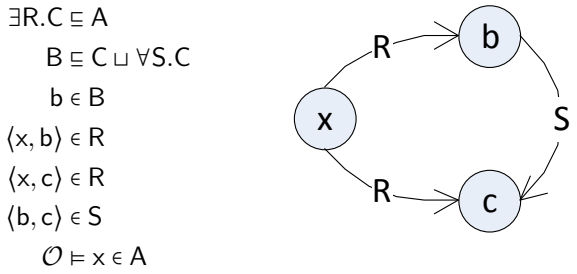


Figure 4.3: Example for inferences by individual classification

model inferences through reasoning. The example in Figure 4.3 shows an ontology consisting of TBox and ABox which implies that $x \in A$. However, it is not possible to express the same information solely in the TBox.

The need for a hybrid integration can be shown at the Patient Chronicle Model case study. The workflow presented in Chapter 4.2 shows that the Clinical Problem Glimpse has to change its structure since new properties emerge during the classification. This is not feasible to implement using a direct integration. Although, this workflow is feasible using an indirect integration, this approach has the shortcoming of an awkward implementation for the Java developers since the model has no domain specific API. Consequently, a hybrid approach is the most suited approach.

The main goal of a hybrid modelling approach is the preservation of important features of the integrated modelling formalisms. In the case of OWL this means that reasoning services like consistency checking, complex class classification, individual classification, and individual property inferences as well as post-coordination, which implies the support of multiple typing with complex classes, should be supported. The restrictions on the modelling imposed by the integration framework should be as minimal as possible so that domain experts can still use flexible means for modelling the domain. Important features of Java which should be preserved are type safety, encapsulation, and the expressiveness concerning behaviour. Furthermore, the software developers should be enabled to efficiently develop complex ontology-based applications, or, in other words, the framework should be easy to use.

The requirement for a flexible integration, Requirement 2, demands that the *integration semantics* has to be adaptable to the needs of the use case. Integration semantics refers to the interpretation of the OWL model within Java using OO model elements. Because the OO and ontological modelling formalism are not congruent concerning their expressiveness, the integration semantics often has to impose some kind of sanctioning mechanism. For example, it can be interesting to restrict the accessible OWL properties in the OO model or derive a static class structure from an OWL class description. However, such sanctioning mechanisms are highly domain specific and depend on the requirements of a concrete application which should be implemented. Therefore, it is important, that the integration semantics can be adapted to the specific needs of a development project.

The Requirement 3 aims at the introduction of a clear separation of concerns, i.e., the separation of code for the integration the OWL model into the OO model from domain model code containing the business logic. This makes the model easier to understand and allows for the separation and specialisation of developers: on the one hand, there can be Java developers working with the Java model and the hybrid classes without worrying about the integrated ontology and, on the other hand, there can be ontology integration developers who define the integration semantics. The targeted separation can be accomplished through a declarative definition of the integration or, more precisely, the declarative definition of hybrid classes. An example for this which makes use of Java annotations has already been shown in Listing 1.1. Furthermore, the utilisation of Java annotations makes the framework very easy to use.

Besides these requirements, there are no other needs to be fulfilled. Although performance considerations are important for the productive use of a framework like Mooop, this work concentrates on the features of a generic integration concept and, therefore, neglects the efficiency of the implementation.

Chapter 5

Concept of Mooop

The three main requirements for a generic integration approach (see Chapter 4.3) reveal two distinct concerns: on the one hand, there is the translation of an OWL model into Java which ought to be flexible and preservative concerning important OWL features. On the other hand, there is the integration into the application's Java model which ought to be declarative and preservative concerning important Java features. Furthermore, as a hybrid integration approach, Mooop uses special Java model classes, called *hybrid classes*, to represent and integrate OWL model classes and properties.

The Mooop concept reflects the requirements by introducing three conceptual layers as shown in Figure 5.1:

1. The *OwlFrame* is a Java component which represents information encoded in an OWL model. It is a container for the indirect integration of an OWL individual, which offers a generic, domain neutral representation of ontological information in Java. The combination of an OwlFrame and an OWL individual is called *straddling object* because it straddles both models.
2. The *mapping* is a flexible link between the OWL model and an OwlFrame. It defines how the OwlFrame represents the ontological information, i.e., the integration semantics. The flexibility of the mapping is necessary for customising and adapting it to the specific needs of an application.
3. The *binding* is a declarative link between an OwlFrame and the Java model. It enables the definition of *hybrid classes* which represent OWL classes in the Java model. Their instances, namely *hybrid objects*, allow the domain specific access to the information of an OwlFrame. Hybrid classes are defined in a declarative manner using Java annotations, thereby, separating the integration logic from the business logic of the application.

A basic principle of Mooop is to allow the adaptation of the integration to the specific requirements of a project. The three-layered architecture facilitates this by providing a separation of concerns, which, consequently, allows for a developer specialisation, and allows for customizing the binding and mapping. This chapter describes the three conceptual layers in detail.

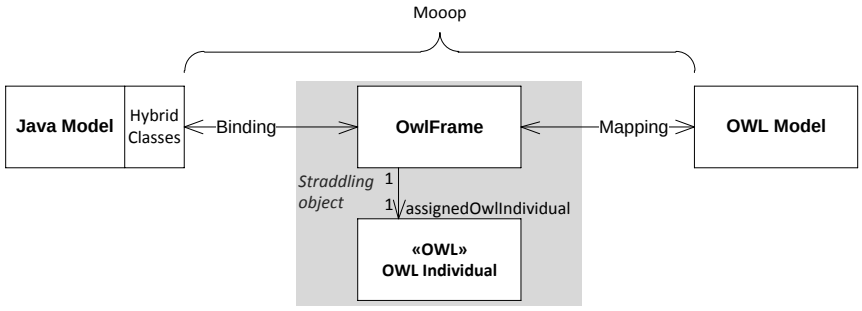


Figure 5.1: The three conceptual layers of Mooop.

5.1 OwlFrame

Figure 5.2 depicts the OwlFrame as a frame-like, generic representation of information about an OWL individual. It has an injective assignment to and OWL individual and, therefore, together they form a *straddling object* which resides in Java (the OwlFrame) and OWL (the OWL individual). Notice that the injective assignment does not imply an injective link between Java model objects and OWL individuals (see Chapter 5.3.2).

The frame-like shape of the OwlFrame is inspired by the representation of an OWL individual in the Manchester syntax for OWL 2 [26]. Frame systems are an approach for knowledge representation introduced by Minsky [42]: a frame aggregates all knowledge about an object instead of distributing it over several axioms like OWL does [63, p. 140]. Hence, this representation is well suited for OO systems.

The information of an OwlFrame can be represented in both Java and OWL. On the one hand, its frame-like shape is tightly related to the Manchester syntax for OWL 2 and, on the other hand, frames can be easily represented in Java. Thus, this layer of abstraction divides the impedance gap between Java and OWL into two smaller ones: the one between OWL model and OwlFrame and the one between OwlFrame and Java model. Hence, the complexity of the integration is reduced which makes it easier to overcome the impedance mismatch. In the following, the structure and the behaviour of the OwlFrame will be described in more detail.

5.1.1 Structure

The structure of the OwlFrame defines a generic object-oriented representation of ontological data. It is an indirect model (see Chapter 2.2.2), whereby the OwlFrame represents the entity: it is associated to its types and to its properties. The concrete types and properties of an OwlFrame are derived from the knowledge about the assigned OWL individual in the OWL model.

In contrast to a generic indirect model entity, an individual in OWL can have multiple types. Each type can be either an atomic or a complex OWL class. The OwlFrame can also be associated with several OWL classes as its

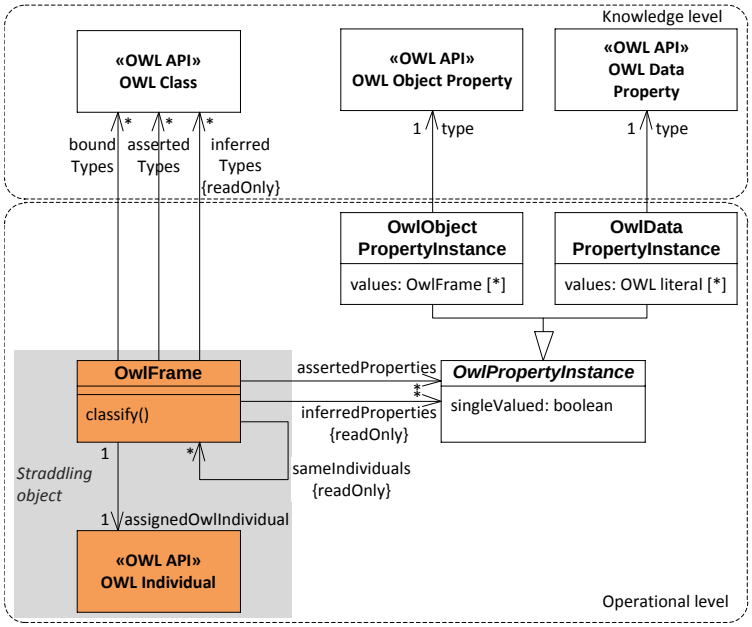


Figure 5.2: The static structure of the OwlFrame.

types. Thereby, the OWL Class can represent both atomic and complex OWL classes. Furthermore, there is not only one kind of type but three:

Asserted types are the explicit types of the OwlFrame. They can be queried and manipulated.

Inferred types are inferred from the explicit knowledge about the OwlFrame. Usually, they are the result of a classification of the assigned OWL individual. They can be queried but they can not be manipulated.

Bound types are special explicit types of the OwlFrame. They can be queried and manipulated.

The concrete semantics of theses different types is, however, defined by the mapping. The bound types are important for the binding layer introduced in Chapter 5.3 by offering a type safety for hybrid objects. Thus, the bound types are usually not manipulated by the application using the Mooop framework but by the binding layer. The complex typing facility of the OwlFrame enables complex post-coordination (see Chapter 2.3) because it allows multiple typing of OWL individuals using complex OWL classes.

Another difference between the Mooop concept and an indirect model is that the types of an OwlFrame do not define its properties which reflects the lax object conformance in OWL. Hence, in Mooop it is possible to change property values and, also, to define new properties of an OwlFrame. There are two different types of properties:

Asserted properties are the explicit properties of the OwlFrame. They can be queried and manipulated.

Inferred properties are inferred from the explicit knowledge about the OwlFrame. Usually, they are the result of a classification of the assigned OWL individual. They can be queried but they can not be manipulated.

The properties of an OwlFrame can be of one out of two types: OwlObjectPropertyInstance or OwlDataPropertyInstance. The former represents all values for an OWL object property, i.e., OWL individuals, in form of OwlFrame objects. The latter represents all values for an OWL data property, i.e., OWL literals, as OWL Literal objects. The type of an OwlObjectPropertyInstance or an OwlDataPropertyInstance is an OWL Object Property or an OWL Data Property, respectively. The distinction between relationships among objects (OwlObjectPropertyInstance) and data values (OwlDataPropertyInstance) is a common modelling feature and recommended for the usage of an AOM as well. Furthermore, while doing research on AOMs, Yoder et al. found out that, "while few language designers seem to feel the need to represent these relationships, most designers of systems with Adaptive Object-Models do."[75]

In OO systems a property value can be single- or multivalued. In parallel, the OwlPropertyInstance has an attribute isSingleValued which controls several checks triggered by manipulations. These checks ensure that the size of the value array of a single-valued property instance can not be larger than one. This allows three states for a single-valued property instance: (1) no-value if the value array is empty, (2) null-value if the value array contains a null, or (3) normal-value if the value array contains a normal value. The interpretation and translation of these states into the OWL model and Java model is the duty of the mapping.

Both the types and the properties of the OwlFrame are separated into asserted and inferred ones because of their different semantics in OWL. While the asserted types and properties define the explicit knowledge about an OWL individual, the inferred types and properties are implied knowledge derived from the entire OWL model. Hence, the inferred information can indirectly change with every manipulation of the model, but the asserted information only by direct manipulation. As this distinction does not exist in Java, it has to be simulated: asserted types and properties are normal OO properties and types which can be read and written, and inferred types and properties can be seen as the result of an OO query which is read-only.

The absence of a separation of asserted and inferred knowledge can lead to the problem of unintended restrictions. Imagine the following scenario in the pizza configurator case study: a user tops a new pizza (Pizza) solely with Ham. From the axioms shown in Figure 5.3, the system can now infer that the pizza is of type MildPizza. This is interesting information for the system user. However, if the inferred type MildPizza would be mixed with the asserted types, it would mean that the user is trying to create a MildPizza. Therefore, the adding of a HotTopping like JalapenoPepperTopping as the next step would result in an inconsistent OWL model. This unwanted behaviour can be avoided by the clean separation between asserted and inferred information.

Indirect models implicitly assume a UNA, i.e., each entity is different form all the others. However, since OWL does not assume this, it can be the case that two OWL individuals are (explicitly or implicitly) known to be the same.

$$
\begin{aligned}
\mathsf{Ham} &\sqsubseteq \mathsf{PizzaTopping} \\
\mathsf{Ham} &\sqsubseteq \exists \mathsf{hasSpiciness.Mild} \\
\mathsf{MildTopping} &= \mathsf{PizzaTopping} \sqcap \exists \mathsf{hasSpiciness.Mild} \\
\mathsf{MildPizza} &= \mathsf{Pizza} \sqcap \exists \mathsf{hasTopping.PizzaTopping} \sqcap \forall \mathsf{hasTopping.MildTopping}
\end{aligned}
$$

Figure 5.3: Sample axioms from the pizza configurator exemplifying unintended restrictions.

$$
\begin{aligned}
\mathsf{PizzaMargherita} &= \mathsf{Pizza} \\
&\quad \exists \mathsf{hasTopping.Mozzarella} \sqcap \exists \mathsf{hasTopping.Tomato} \\
\mathsf{pizza} &\in \mathsf{Pizza} \\
\mathsf{mozzarella} &\in \mathsf{Mozzarella} \\
\mathsf{tomato} &\in \mathsf{Tomato} \\
\langle \mathsf{pizza}, \mathsf{mozzarella} \rangle &\in \mathsf{hasTopping} \\
\langle \mathsf{pizza}, \mathsf{tomato} \rangle &\in \mathsf{hasTopping}
\end{aligned}
$$

Figure 5.4: Example ontology defining a class PizzaMargherita and an OWL individual pizza.

Consequently, an OwlFrame is associated to other OwlFrames representing different OWL individuals which can be asserted or inferred to be the same. For instance, imagine a pizza has a unique number which identifies exactly one individual pizza. This ID would be modelled as a functional OWL data property in the ontology. If two distinct OWL individuals have the same ID then they are inferred to be actually the same. However, sameIndividuals can only be queried but not manipulated. This restriction aligns the OwlFrame with the OO principle of a UNA while allowing to gather as much information from the OWL model as possible.

OWL allows the expression of positive and negative assertions about the properties and same individuals of an OWL individual. Due to the CWA, OOP does not explicitly express negative properties because they are implicit. In order to reduce the complexity of the overall integration, the OwlFrame is aligned with OOP and does not distinguish between them. If it is necessary to express negative assertions, these have to be integrated by the mapping (see Chapter 5.2) as, e.g., special positive properties.

In order to exemplify the OwlFrame structure, Figure 5.4 shows an example OWL ontology defining a pizza Margherita and Figure 5.5 depicts a possible representation of an OWL individual pizza by an OwlFrame. However, this is just one possibility since the properties and types of an OwlFrame depend on the integration semantics defined by the mapping.

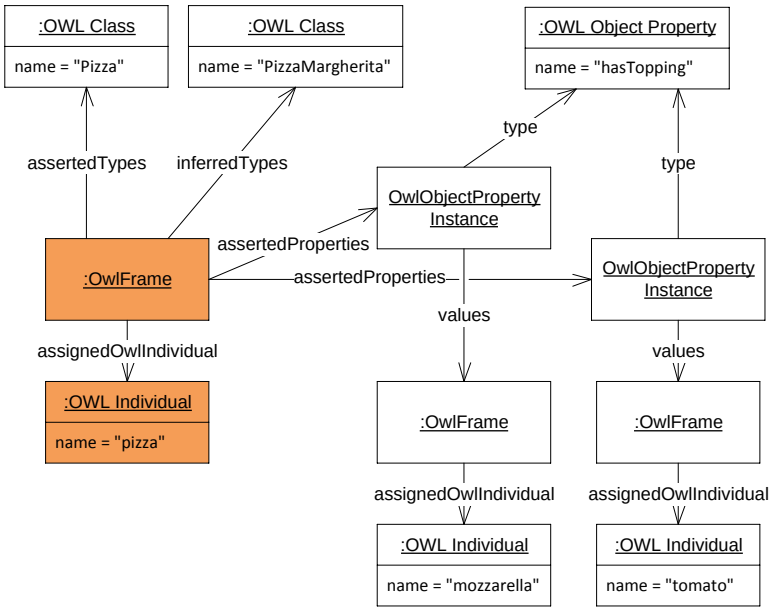


Figure 5.5: The runtime object model for the OwlFrame representing the OWL individual pizza.

5.1.2 Behaviour

OWL can not express behaviour. Therefore, the OwlFrame does not have to represent any OWL behaviour. However, an OwlFrame offers an OO behaviour in form of:

Manipulations of bound types, asserted types and asserted properties in form of additions and removals through methods like addAssertedType(). These methods are delegating the calls to the mapping (see Chapter 5.2). For clarity reasons, they are not depicted in Figure 5.2.

Classification of the OwlFrame, i.e., classification of the assigned OWL individual by a reasoner and, thus, inferring new knowledge. These new information will usually be represented as inferred types and inferred properties. The classification is triggered by the classify() method.

Reasoning in the OWL model, i.e., the invocation of selected general reasoning services in the ontology. This service does not manipulate the OWL model but only gathers furthers knowledge. It can be used to identify subclass relationships between OWL classes.

The OwlFrame does only offer an interface for the client to invoke this behaviour. The semantics of it, i.e., the results and modifications in the OWL model, are defined by the mapping as shown in Chapter 5.2.

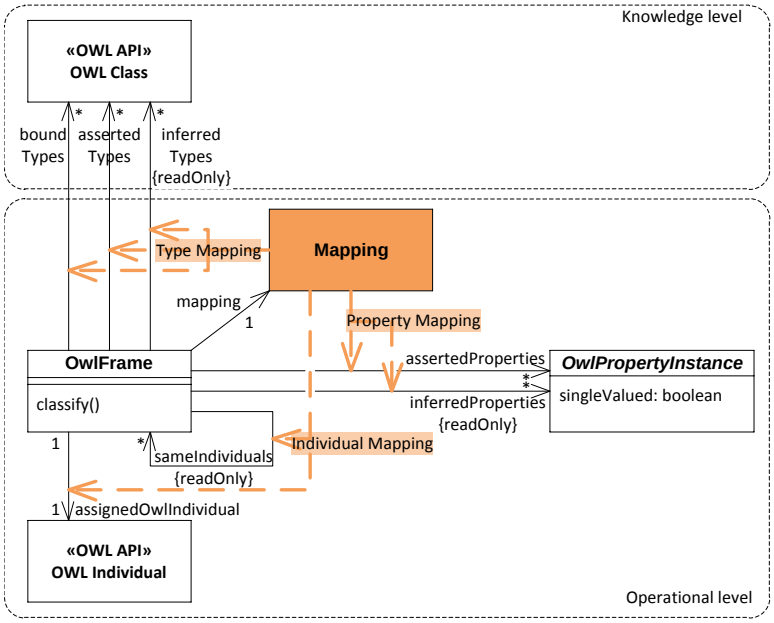


Figure 5.6: The mapping and its sub-mappings in the context of the OwlFrame.

5.2 Mapping

The mapping links the knowledge of the OWL model to the features of an OwlFrame. The connection between mapping and OwlFrame is shown in Figure 5.6. The mapping is used by the OwlFrame in order to access the OWL model. Hence, the OwlFrame can be seen as a data container and the mapping as the definition of the semantics of its features, i.e., the *integration semantics*.

An OwlFrame is associated to exactly one mapping. This separation of data container and logic is an instance of the state pattern [18]. However, in contrast to the original pattern, the reason for the separation of state and strategy in case of the Mooop framework is to facilitate customization of the mapping by the users of the framework. Since the mapping is a stateless collection of methods, it is easy to combine several different mappings to one. This flexibility is necessary because the integration semantics depends on the project specific OWL modelling guidelines which include, e.g., a custom sanctioning mechanism.

In Mooop, all OwlFrames use the same mapping and, thus, there is a single integration semantics for all hybrid classes. However, one can also think of an approach which defines a mapping for each hybrid class. This domain-model specific integration allows, e.g., that the properties of pizzas are represented differently from the properties of pizza toppings. However, since OWL allows multiple typing, this approach can lead to illegal object states: If one hybrid object a manipulates an OWL individual o in a way which is not allowed by the integration semantics of another hybrid object b which also represents o, then the state of o is illegal for b. In contrast to that, the Mooop approach allows a domain specific integration without coupling to the concrete Java domain

Symbol	Meaning
$\mathcal{O}$	OWL ontology
i	OWL individual (in both Java and OWL)
l	OWL literal (in both Java and OWL)
t	OWL class (in both Java and OWL)
r	OWL object property (in both Java and OWL)
u	OWL data property (in both Java and OWL)
A	URI used to identify OWL individuals
o	OwlFrame
$i(o)$	OWL individual assigned to the OwlFrame o. This function can also be used in DL axioms instead of some i.
V^O	Set of OwlFrame objects
V^D	Set of OWL Literal objects
$[r = V^O]$	OwlObjectPropertyInstance with the type r and the values V^O
$[u = V^D]$	OwlDataPropertyInstance with the type u and the values V^D

Table 5.1: The symbols for defining the mapping.

model. The drawback of this is that specialized semantics for OWL classes like the one mentioned above are more difficult to express. However, the need for such constructs is probably often a sign for an incoherent modelling guideline which should be avoided.

The mapping distinguishes 4 sub-mappings:

Type mapping defines the asserted, inferred, and bound types of an OwlFrame. Furthermore, it defines valid manipulations of the asserted and bound types.

Property mapping defines the asserted and inferred properties of an OwlFrame. Furthermore, it defines valid manipulations of the asserted properties.

Individual mapping defines the sameIndividuals of an OwlFrame. Furthermore, it defines the classification process, and the creation and deletion of the assigned OWL individual.

Reasoning mapping provides an access to reasoning services for the ontology. This comprises a subclass check for OWL classes and a sub-property check for both OWL object properties and OWL data properties.

Mooop provides some basic implementations for the sub-mappings which can be used out of the box. The following introduction of the sub-mappings is accompanied by an explanation of the basic semantics as well as some ideas for advanced semantics. Thereby, the symbols defined in Table 5.1 will be used. Notice that each symbol can be decorated with subscripts, e.g., i_1. In particular, for o, an OwlFrame, $i(o)$ is used to refer to the assigned OWL individual.

5.2.1 Type Mapping

The type mapping defines all type related structural and behavioural features of an OwlFrame. These are the asserted, inferred, and bound types, as well as valid manipulations of the asserted and bound types. Inferred types are read-only. The types are represented by the Java class OWL Class which indirectly integrates OWL classes from the OWL model. Hence, the types of an OwlFrame can represent both atomic and complex classes.

The type mapping has to provide a specific contract, i.e., an interface, in order to be used by the OwlFrame. This interface comprises the following functions and the intended results:

$\mathbf{getTypes_b}(o)$ yields the set of bound types of the OwlFrame o.

$\mathbf{getTypes_a}(o)$ yields the set of asserted types of the OwlFrame o.

$\mathbf{getTypes_i}(o)$ yields the set of inferred types of the OwlFrame o.

$\mathbf{addType_b}(o, t)$ adds the OWL class t to the bound types of o.

$\mathbf{removeType_b}(o, t)$ removes the OWL class t from the bound types of o.

$\mathbf{addType_a}(o, t)$ adds the OWL class t to the asserted types of o.

$\mathbf{removeType_a}(o, t)$ removes the OWL class t from the asserted types of o.

Notice that these descriptions are only intended semantics of the functions. The actual semantics is defined by a particular mapping and can be customized. In the following, we outline the semantics for a basic mapping.

The basic semantics of the set of asserted types $\text{getTypes}_\text{a}(o)$ of an OwlFrame o is:

$$\text{getTypes}_\text{a}(o) = \{t | (i(o) \in t) \in \mathcal{O}\}.$$

The asserted types are constantly written to the value of the attribute assertedTypes of the OwlFrame o.

Accordingly, the basic semantics of the set of inferred types $\text{getTypes}_\text{i}(o)$ of an OwlFrame o is:

$$\text{getTypes}_\text{i}(o) = \{t | \mathcal{O} \vDash i(o) \in t \text{ and } t \notin \text{getTypes}_\text{a}(o)\}.$$

Notice that the asserted and inferred types are disjoint. The inferred types are not constantly written to the value of the attribute inferredTypes of the OwlFrame o. Instead this is usually done during classification (see Chapter 5.2.3).

The bound types can be seen as a set of special asserted types in order to offer type safety for hybrid objects (see Chapter 5.3). Although they are used by the binding, the mapping defines the meaning of the bound types because they have great influence on the valid manipulations of the asserted types and properties.

In the basic mapping, the manipulations of the bound types are not restricted and can be performed through the following methods which have the described effects:

$$(\mathsf{Vehicle} \sqsubseteq \exists \mathsf{hasDriver}.\top) \in \mathcal{O}$$
$$\mathcal{O} \not\models \mathsf{Vehicle} \sqsubseteq \mathsf{Pizza}$$

Figure 5.7: Example ontology defining a Vehicle.

addType$_\mathbf{b}$ (o,t)**:** $t \in \text{getTypes}_\text{b}(o)$ and $t \in \text{getTypes}_\text{a}(o)$.

removeType$_\mathbf{b}$ (o,t)**:** $t \notin \text{getTypes}_\text{b}(o)$.

The customizable mapping allows for different types of type safety. The *weak type safety* is the most trivial form. It prevents the removal of a bound type from the asserted types. Hence, it ensures that an OwlFrame is at least an instance of its bound types. Therefore, the manipulation methods yield the following effects:

addType$_\mathbf{a}$ (o,t)**:** $t \in \text{getTypes}_\text{a}(o)$ and $(i(o) \in t) \in \mathcal{O}$. The latter part of the conjunction means that the axiom $(i \in t)$, whereby $i = i(o)$, is added to the ontology.

removeType$_\mathbf{a}$ (o,t)**:** if $t \notin \text{getTypes}_\text{b}(o)$, then $t \notin \text{getTypes}_\text{a}(o)$ and $(i(o) \in t) \notin \mathcal{O}$; otherwise an exception is thrown.

The *strong type safety* is derived from the weak type safety. It additionally ensures that the asserted types are only subclasses or super classes of the bound types. Therefore, the following methods have to be redefined:

addType$_\mathbf{a}$ (o,t)**:** if there is a bound type $t_x \in \text{getTypes}_\text{b}(o)$ such that $\mathcal{O} \models t \sqsubseteq t_x$ or $\mathcal{O} \models t_x \sqsubseteq t$, then $t \in \text{getTypes}_\text{a}(o)$ and $(i(o) \in t) \in \mathcal{O}$; otherwise an exception is thrown.

removeType$_\mathbf{b}$ (o,t)**:** $t \notin \text{getTypes}_\text{b}(o)$, and for all the asserted types $t_x \in \text{getTypes}_\text{a}(o)$ there exists a type $t_y \in \text{getTypes}_\text{b}(o)$ such that $\mathcal{O} \models t_x \sqsubseteq t_y$ or $\mathcal{O} \models t_y \sqsubseteq t_x$.

This restriction seems very reasonable for a hybrid integration. In fact, the Core Model-Builder (see Chapter 3.3) implements a similar definition. Hence, Mooop uses the strong type safety as the default.

One can think of even more restrictive semantics for type safety, e.g., that not only the asserted types have to conform the strong type safety property but also the inferred types. This can make a difference in the following example: an OwlFrame o should represent a pizza and, thus, $\text{getTypes}_\text{b}(o) = \{\mathsf{Pizza}\}$. If you assume the ontology from Figure 5.7, then setting the value of the OWL property hasDriver for o to some OWL individual o_x (see Chapter 5.2.2) would be intercepted and forbidden because then $\mathcal{O} \models i(o) \in \mathsf{Vehicle}$. However, these semantics have not been researched in more detail for this work.

Further extensions of the basic type mapping could make use of OWL annotations. For instance, types could be hidden or not allowed to be asserted based on annotations. This makes it clear that the concrete definition of a mapping

is coupled with the concrete OWL modelling guideline. However, the benefit of such advanced guidelines is not in the scope of this work.

5.2.2 Property Mapping

The property mapping defines all property related structural and behavioural features of an OwlFrame. These are the asserted and inferred properties, and valid manipulation of the asserted properties. In parallel to the inferred types, the inferred properties are read-only. A property can be either an OwlObjectPropertyInstance or an OwlDataPropertyInstance. The former represents all values for an OWL object property, i.e., OWL individuals, in form of OwlFrame objects. The latter represents all values for an OWL data property, i.e., OWL literals, as OWL Literal objects.

The property mapping has to provide a specific contract, i.e., an interface in order to be used by the OwlFrame. This interface comprises the following functions and the intended results:

getProperties$_\mathbf{a}$ (o) yields the set of asserted properties, i.e., OwlObjectPropertyInstance objects and OwlDataPropertyInstance objects, of the OwlFrame o.

getProperties$_\mathbf{i}$ (o) yields the set of inferred properties, i.e., OwlObjectPropertyInstance objects and OwlDataPropertyInstance objects, of the OwlFrame o.

addProperty$_\mathbf{a}$ (o, r, o_x) adds the value o_x for the OWL object property r to the asserted properties of o.

addProperty$_\mathbf{a}$ (o, u, l) adds the value l for the OWL data property u to the asserted properties of o.

removeProperty$_\mathbf{a}$ (o, r, o_x) removes the value o_x for the OWL object property r from the asserted properties of o.

removeProperty$_\mathbf{a}$ (o, u, l) removes the value l for the OWL data property u from the asserted properties of o.

Notice that these descriptions are only intended semantics of the functions. The actual semantics is defined by a particular mapping and can be customized. In the following, we outline the semantics for a basic mapping.

The basic semantics for the set of asserted properties $\text{getProperties}_\text{a}(o)$ of an OwlFrame o is:

$$\begin{aligned}\text{getProperties}_\text{a}(o) = &\{[r \;=\; V^O] | V^O = \{o_n | (\langle i(o), i(o_n)\rangle \in r) \in \mathcal{O}\} \text{ and } V^O \neq \varnothing\} \\ &\cup \{[u \;=\; V^D] | V^D = \{l | (\langle i(o), l\rangle \in u) \in \mathcal{O}\} \text{ and } V^D \neq \varnothing\}.\end{aligned}$$

The asserted properties are constantly written to the value of the attribute assertedProperties of the OwlFrame o.

The basic semantics for the set of inferred properties $\text{getProperties}_\text{i}(o)$ of an

OwlFrame o is:

$$\begin{aligned}\text{getProperties}_{\text{i}}(o) = \{&[r \;=\; V^O] | V^O = \{o_n | \mathcal{O} \vDash \langle i(o), i(o_n)\rangle \in r \\ &\text{and, if } [r \;=\; V_x^O] \in \text{getProperties}_{\text{a}}(o) \text{ then } o_n \notin V_x^O\} \\ &\text{and } V^O \neq \varnothing\} \\ &\cup \{[u \;=\; V^D] | V^D = \{l | (\langle i(o), l\rangle \in u_x) \in \mathcal{O} \text{ and } \mathcal{O} \vDash u_x \sqsubseteq u \\ &\text{and, if } [u \;=\; V_x^D] \in \text{getProperties}_{\text{a}}(o) \text{ then } l \notin V_x^D\} \\ &\text{and } V^D \neq \varnothing\}.\end{aligned}$$

Notice that the set of the OwlDataPropertyInstance objects is not determined by using $\mathcal{O} \vDash \langle i(o), l\rangle \in u$ because the used OWL reasoner framework does not provide this feature. The asserted and inferred properties are disjoint. The inferred properties are not constantly written to the value of the attribute inferredProperties of the OwlFrame o. Instead this is usually done during classification (see Chapter 5.2.3).

In OOP, there is generally the distinction between single and multiple valued properties. In OWL, such a concept does not directly exist. Although OWL knows functional properties and complex class definitions with exactly and maximal cardinality restrictions on OWL properties, it is hard to actually make sure that a property has at most one value. For instance, assume the OWL object property hasCustomer which is functional, and the OWL individuals pizza, john, and johann. In this setting, it is allowed to state $\langle \text{pizza}, \text{john}\rangle \in \text{hasCustomer}$ and $\langle \text{pizza}, \text{johann}\rangle \in \text{hasCustomer}$. The resulting inference is $\mathcal{O} \vDash \text{john} = \text{johann}$. However, in the Mooop concept, both OWL individuals would be represented by separate OwlFrames. Therefore, the OwlFrame o with $i(o) = \text{pizza}$ would have two values for the property hasCustomer. The only way of ensuring single valued properties is in combination with an overall UNA in the OWL model. However, a UNA should be imposed by the individual mapping (see Chapter 5.2.3) and, hence, its implementation is customisable. So, it is not easy to find out, whether there is a general UNA in the model[1]. Hence, in the basic mapping, we decided for a practical solution which is easy to implement: every functional property is seen as a single value property. If the framework comes across a functional property with several values, an exception is thrown. In case this is not suitable for a domain, this behaviour can be customised. Notice that the UNA is not necessary for data properties since an implicit UNA is always assumed for them.

The basic semantics does not restrict the valid manipulations on the asserted properties, i.e., values for arbitrary OWL properties can be added and removed from the OwlFrame. The following enumeration describes the effects of these methods.

addProperty$_{\text{a}}(o, r, o_x)$: there is a $[r \;=\; V^O] \in \text{getProperties}_{\text{a}}(o)$ and $o_x \in V^O$ and $(\langle i(o), i(o_x)\rangle \in r) \in \mathcal{O}$. The third part of the conjunction means that the axiom $(\langle i, i_x\rangle \in r)$, whereby $i = i(o)$ and $i_x = i(o_x)$, is added to the ontology.

addProperty$_{\text{a}}(o, u, l)$: there is a $[u \;=\; V^D] \in \text{getProperties}_{\text{a}}(o)$ and $l \in V^D$ and $(\langle i(o), l\rangle \in u) \in \mathcal{O}$.

[1] Notice that it has to be ensured that all current OWL individuals are distinct from each other but also future ones.

removeProperty$_{\mathbf{a}}$ (o, r, o_x)**:** if there is a $[r = V^O] \in \text{getProperties}_{\text{a}}(o)$, then $o_x \notin V^O$ and $V^O \neq \varnothing$; and $(\langle i(o), i(o_x)\rangle \in r) \notin \mathcal{O}$.

removeProperty$_{\mathbf{a}}$ (o, u, l)**:** if there is a $[u = V^D] \in \text{getProperties}_{\text{a}}(o)$, then $l \notin V^D$ and $V^D \neq \varnothing$; and $(\langle i(o), l\rangle \in r) \notin \mathcal{O}$.

The property mapping is very likely to be customized. Besides hidden or forbidden properties based on OWL annotations, one can think of imitating a strict object conformance by some kind of sanctioning. Thereby, the inferred properties would hold default or null values for all properties for which no value was asserted but is inferred to be existent. Hence, the asserted and inferred properties of an OwlFrame together strictly conform to an OWL class definition. Consequently, the setting of values for properties which are not defined by the sanctioning is denied. Another extension one can define is that null values for properties are interpreted as negative property assertions which can not be set directly (see Chapter 5.1.1). An extension which is used in the pizza configurator case study is the definition of a covering axiom over the properties of an OwlFrame. If a pizza has the toppings $\mathsf{topping}_1 \ldots \mathsf{topping}_n$ then the OwlFrame o for the pizza has an asserted type $\forall \mathsf{hasTopping}.\{\mathsf{topping}_1, \ldots, \mathsf{topping}_n\}$ which defines that the OwlFrame has no more than the asserted toppings.

5.2.3 Individual Mapping

The individual mapping defines all individual related structural and behavioural features of an OwlFrame. These are the sameIndividuals as well as controlling the classification process of an OwlFrame and the life cycle of the assigned OWL individual.

The individual mapping has to provide a specific contract, i.e., an interface in order to be used by the OwlFrame. This interface comprises the following functions and the intended results:

getSameIndividuals(o) yields the set of OwlFrames o_x which assigned OWL individual $i(o_x)$ is inferred to be identical to $i(o)$.

create(o, A) creates a new OWL individual i with the URI prefix A, adds it to the ontology, and assigns it to o.

load(o, A) loads the OWL individual with the URI A from the ontology and assigns it to o.

classify(o) classifies the OWL individual $i(o)$ using a reasoner and, thus, infers new knowledge. The new information will be usually represented as the inferred types, the inferred properties, and the same individuals of o.

delete(o) removes the OWL individual $i(o)$ from the ontology.

Notice that these descriptions are only intended semantics of the functions. The actual semantics is defined by a particular mapping and can be project specific. However, in the following, we outline the semantics for a basic mapping.

The basic semantics for the set of same individuals getSameIndividuals(o) of an OwlFrame o is a simple representation of the knowledge in the OWL model:

$$\text{getSameIndividuals}(o) = \{o_x | \mathcal{O} \vDash i(o) = i(o_x)\} \setminus \{o\}.$$

Notice that the same individuals can be asserted in the OWL model or implied by it. An OwlFrame is not contained in its own set of same individuals. The values of this association are again OwlFrames. The same individual are not constantly synchronized with the knowledge in the OWL model, i.e., after every modification of the ontology. Instead this is usually done during classification.

When a new OWL individual should be created with a given URI prefix, then the mapping adds a unique number to that URI and subsequently creates an OWL individual with this modified URI:

create(o, A)**:** $i(o)$ has the URI A_x and $i(o) \in \mathcal{O}$ and $A_x = A + n$ (i.e., A concatenated with n) and $n \in \mathbb{N}$ and there is no $i_x \in \mathcal{O}'$ (i.e., $\mathcal{O}$ before the execution of the function) such that i_x has the URI A_x.

When an OWL individual should be loaded, then the mapping checks whether an OWL individual with the URI already exists in the OWL model and loads it:

load(o, A)**:** if there exists a $i \in \mathcal{O}'$ such that i has the URI A, then $i(o) = i$; otherwise an exception is thrown.

Furthermore, a unique name assumption is enforced for all OWL individuals. This reflects the normal semantics of the OO model and is a reasonable default. Therefore, the following invariant of the ontology holds at any time:

$$\text{for all } i_x \in \mathcal{O}, i_y \in \mathcal{O} \text{ holds } \mathcal{O} \vDash i_x \neq i_y.$$

The classification of an OwlFrame o has to be explicitly triggered using classify(o). This enables *complex manipulations* of o, which can involve several changes on the types and properties of o. In order for such a complex manipulation to be valid, it has to start in a valid OO and OWL model and it has to end in a valid OO and OWL model. However, the intermediate states, i.e., the states after each single type or property manipulation, do not have to be valid. This can be compared to transactions in database systems: they allow the aggregation of small data manipulations which by themselves are illegal, i.e., violate an integrity rule, but in combination are legal. Hence, a complex manipulation together with an automatic classification for minor manipulations of o can cause classification errors.

The basic classification process proceeds in two steps: first, the assigned OWL individual is classified in the OWL model using a reasoner and, second, the inferred types, inferred properties, and same individuals are written from the OWL model to the Java model, i.e., the knowledge from the OWL model is represented in the OwlFrame according to the mapping. Thus, the basic semantics of classify(o) yields the following effect:

classify(o)**:** the value of the attribute inferredTypes of the OwlFrame o is equal to $\text{getTypes}_\text{i}(o)$, the value of the attribute inferredProperties of the OwlFrame o is equal to $\text{getProperties}_\text{i}(o)$, and the value of the attribute sameIndividuals of the OwlFrame o is equal to of getSameIndividuals(o).

Notice that the basic semantics constantly writes the values of the asserted types and properties to the respective attributes of the OwlFrame. If a custom mapping does not do this, a preparation phase is necessary which synchronizes the asserted types and properties from the Java model to the OWL model. For instance, assume that a lazy mapping does not add the assertion $i(o) \in t$ to the ontology when $\text{addType}_\text{a}(o, t)$ is called, then assertions like this have to be added to the ontology during the preparation phase. In case that the classification fails, caused by an inconsistency in the OWL model, an exception is thrown. This exception can convey further debugging information concerning the failure gathered from an OWL debugging facility like the one explained in [55]. The basic individual mapping performs the classification on the complete model, i.e., all OwlFrames are classified together. Thus, the Java model is completely synchronized with the OWL model which guarantees that the Java model is consistent. Otherwise, it could happen that, e.g., an OwlFrame o has o_x as a same individual but not vice versa.

The most interesting extension of the classification mechanism is the introduction of some resolution mechanism for inconsistencies. This seems to be highly domain specific and dependant on the modelling guideline since the resolution mechanism has to manipulate the OWL model in some sensible way. Therefore, a detailed research on this enhancement is out of scope of this work.

The deletion of an OwlFrame removes all assertion axioms related to its assigned OWL individual and invalidates the OwlFrame, i.e., removes the assignment between OwlFrame and OWL individual. Thus, the basic semantics for delete(o) yields the following effect:

delete(o)**:** $i(o)$ is not defined and $i(o') \notin \mathcal{O}$ (o' means the OwlFrame before the execution of the method). This implies that there is no axiom (TBox or ABox) in $\mathcal{O}$ which references $i(o')$.

5.2.4 Reasoning Mapping

The reasoning mapping provides a generic access to a classification reasoning service. It allows to check for a subclass or subproperty relation between two OWL classes or OWL properties, respectively. Hence, the reasoning mapping provides further knowledge for the later introduced binding about the OWL model. This information can be used for, e.g., computing the most specific types of an OwlFrame. In contrast to the other three sub-mappings, the reasoning mapping is not used by the OwlFrame but solely by the binding. However, the binding accesses the reasoning mapping through an interface provided by the OwlFrame.

The reasoning mapping has to provide a specific contract, i.e., an interface. This interface comprises the following functions with the described intended result:

isSubType(t, t_s) yields whether t is a subclass of t_s in the ontology.

isSubProperty(r, r_s) yields whether r is a subproperty of r_s in the ontology.

isSubProperty(u, u_s) yields whether u is a subproperty of u_s in the ontology.

Notice that these descriptions are only intended semantics of the functions. The actual semantics is defined by a particular mapping and can be project specific. For instance, it could be necessary to hide an OWL class from the Java model and, thus, a reasoning with this class would cause an exception to be thrown. The reason for omitting an OWL class can be the usage of ontology patterns which introduce solely technical OWL classes.

The following semantics for a basic mapping are directly accessing a reasoning service to query the knowledge:

$$\begin{aligned}
\text{isSubType}(t, t_s) &= \mathsf{true} && \text{if } \mathcal{O} \vDash t \sqsubseteq t_s, \\
&\quad \mathsf{false} && \text{otherwise.} \\
\text{isSubProperty}(r, r_s) &= \mathsf{true} && \text{if } \mathcal{O} \vDash r \sqsubseteq r_s, \\
&\quad \mathsf{false} && \text{otherwise.} \\
\text{isSubProperty}(u, u_s) &= \mathsf{true} && \text{if } \mathcal{O} \vDash u \sqsubseteq u_s, \\
&\quad \mathsf{false} && \text{otherwise.}
\end{aligned}$$

5.3 Binding

The OwlFrame combined with the mapping enables a customizable indirect integration of OWL individuals into Java. The binding allows the declarative definition of a domain specific API for accessing the information in an OwlFrame. Hence, the binding can be seen as the definition of the syntax for the hybrid Mooop integration. The binding in Mooop is twofold:

Structural binding defines an API for accessing the ontological information by binding a specific Java class *hc* to an OWL class t and attributes of *hc* to the information contained in an OwlFrame, e.g., values for an OWL property. The class *hc* is called *hybrid class*.

Runtime binding defines the state and behaviour of the instances of hybrid classes, *hybrid objects*, at runtime. This is necessary since hybrid objects have two identities: an OO object identity defined by themselves and an OWL identity defined by the OwlFrame they are bound to.

Figure 5.8 shows an example which exemplifies the binding. It depicts the relation between hybrid class and hybrid objects and shows that the hybrid class is bound to an OWL class, one attribute is bound to the assertedTypes of the OwlFrame, the method is bound to the method classify() of the OwlFrame, and the hybrid object is bound to an OwlFrame. This example reveals that the bound-to relation can also be seen as a *represents* relation. Hence, the hybrid class represents an OWL class and the bound attribute represents the values of OWL properties.

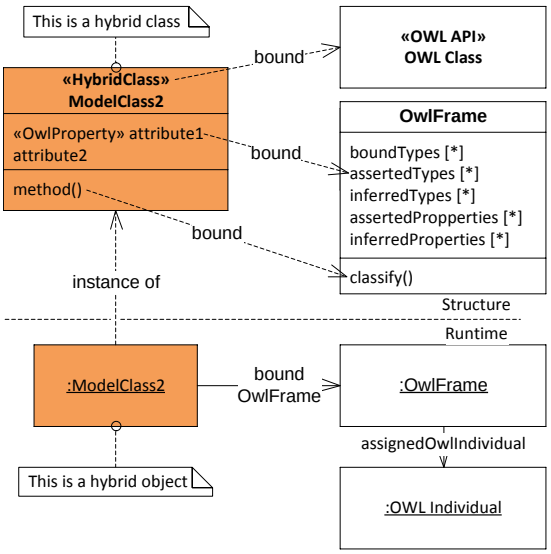


Figure 5.8: The binding in the context of a hybrid class and a hybrid object.

Symbol	Meaning
hc	hybrid class
$t(hc)$	bound OWL class of the hybrid class hc
$[hc \rightarrow o]$	hybrid object of hc which is bound to OwlFrame o

Table 5.2: The additional symbols used for defining the binding.

Figure 5.9 depicts the binding in the context of Mooop. Notice that several hybrid objects can be bound to one OwlFrame object at runtime. A hybrid class directly integrates a specific OWL class into the Java model. Therefore, it usually offers a specific interface for interesting information about OWL individuals of this OWL class. However, OWL allows multiple typing of individuals and, hence, one-to-one binding between hybrid objects and OwlFrames would restrict the integration enormously. Instead, hybrid objects should be seen as *views* as introduced by Harrison and Ossher in [23], offering a domain specific point of view on an OwlFrame. They do not hold any information from the OWL model but instead always query the OwlFrame. As a result, all hybrid objects bound to the same OwlFrame share the same OWL information and are always synchronized with each other.

Besides the symbols already introduced in Table 5.1, the following introduction to the structural and runtime binding uses the additional symbols shown in Table 5.2. Again, they can also be used with subscripts.

5.3.1 Structural Binding

The structural binding defines how a hybrid class in the Java model represents a class from the OWL model as shown in Figure 5.10. The Java class is bound to

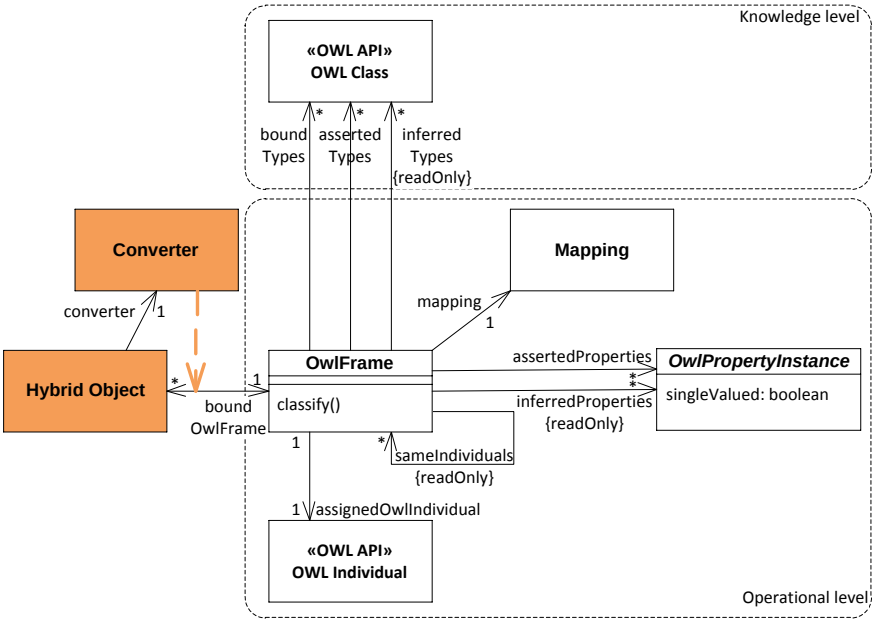


Figure 5.9: The binding in the context of the OwlFrame.

an OWL class, some of its attributes to types and properties of the OwlFrame, and some of its methods to behaviour of the OwlFrame. However, a hybrid class can also have pure OO attributes and methods which are not bound. Furthermore, the structural binding performs a conversion between entities of the OWL model represented through the OwlFrame into entities of the Java model. In the following, these aspects of the structural binding are presented in more detail.

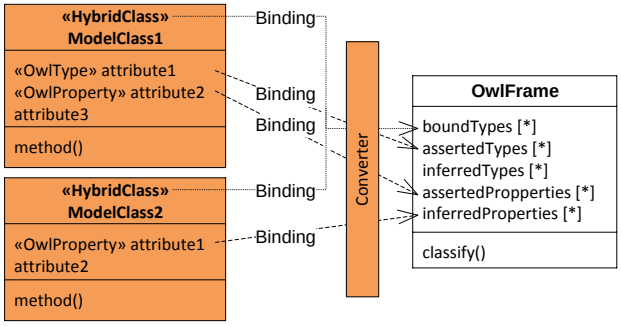


Figure 5.10: The structural binding binds Java classes to OWL classes and Java attributes to properties and types of the OwlFrame.

Class Binding

Mooop allows the annotation of Java classes in order to bind these classes to OWL classes, thereby, creating hybrid classes. The class binding explicitly defines which OWL class the annotated Java class is bound to and can also perform some kind of initialization of the OwlFrame during the binding process. Although Mooop provides a reasonable class binding, it is also possible to develop and use a custom class binding.

Upon instantiation of a hybrid class hc, the hybrid object $[hc \rightarrow o]$ is bound to the OwlFrame o which it should represent. Thereby, the OWL class $t(hc)$ which is bound by hc is added to the bound types of o. The bound types enable a type safety for the hybrid objects. The type safety is defined by the mapping but it should at least ensure that the OWL individual is an instance of the bound types of the OwlFrame. The OWL class $t(hc)$ which is bound by the Java class hc can be either an atomic OWL class or a complex OWL class. This also enables a static post-coordination (see Chapter 2.3).

Figure 5.9 shows that the link between a hybrid object and an OwlFrame object is bidirectional: the hybrid object can access the OwlFrame and the OwlFrame can access the hybrid object. This is necessary because two hybrid classes can be bound to the same OWL class. When both hybrid classes are instantiated for the same OwlFrame then the OwlFrame has only one bound type. When one hybrid object is de-instantiated afterwards (i.e., the hybrid object is deleted and the bound type removed from the OwlFrame; see Chapter 5.3.2) then the OwlFrame has no bound type any more, although one hybrid object is still bound to it. For instance, imaging the hybrid classes hc_1 and hc_2 which are both bound to the OWL class $t = t(hc_1) = t(hc_2)$. Therefore, after creating the hybrid object $[hc_1 \rightarrow o]$, the OwlFrame o has the bound type t. However, after creating $[hc_2 \rightarrow o]$, the bound type of o is still t. Hence, if $[hc_1 \rightarrow o]$ is de-instantiated, the type t will be removed from o. However, this is illegal because $[hc_2 \rightarrow o]$ is still bound to o. This example shows that a direct manipulation of the bound type of the OwlFrame by the class binding is not useful. Therefore, the OwlFrame does only allow the binding to add and remove hybrid objects to the OwlFrame. The OwlFrame queries them for the bound OWL classes of their hybrid classes and, thus, determines its bound types. So, in the previous example, $[hc_1 \rightarrow o]$ and $[hc_2 \rightarrow o]$ do not add their bound types to the OwlFrame but themselves. Afterwards, the OwlFrame asks them for the bound OWL classes and adjusts its bound types. Hence, after the instantiation of both hybrid objects, o has still the bound type t. However, if $[hc_1 \rightarrow o]$ is de-instantiated, it is removed from o. Subsequently, o determines its bound type by asking the remaining hybrid object $[hc_2 \rightarrow o]$ for its bound OWL class and, thus, figures out that it can not remove the bound type t.

The subclass of a Java class is supposed to inherit all properties of the superclass and, according to Liskov's substitution principle [40], objects of the subclass should behave like objects of the superclass. For a hybrid object $[hc \rightarrow o]$, this principle is ensured by Mooop by adding the bound OWL class $t(hc)$ of the instantiated hybrid class hc and all bound OWL classes of the superclasses of hc to the bound types of o. Furthermore, this allows for a static post-coordination through Java inheritance. For instance, take the post-coordination example from Chapter 2.3: assume there is a hybrid class Allergy which is bound to the

OWL class Allergy, and a hybrid class AlmondAllergy which is a subclass of the hybrid class Allergy and bound to the complex OWL class ∃causedBy.Almond. The OwlFrame object o which is bound by a hybrid object [AlmondAllergy → o] has two bound types, Allergy and ∃causedBy.Almond, and, therefore, also the inferred type NutAllergy. Notice that a hybrid class does not necessarily have to be domain specific because a hybrid class which is bound to ⊤ can represent every OWL class.

The basic binding of Mooop is simple and allows to annotate a hybrid class with one bound OWL class, either atomic or complex.

Attribute Binding

The attributes of a hybrid class can be bound to ontological information of the OwlFrame by annotating them. The attribute binding can directly access the bound OwlFrame and, hence, integrate and manipulate the data it holds.

There are numerous ways to represent the information from the OwlFrame as attribute values. Although Mooop already offers a wide range of reasonable bindings, it is important to note that the binding can be customized. Mooop supports this by enabling the easy definition of custom bindings and their integration into the framework as shown in Chapter 6.2.2.

The following basic binding of Mooop offers means to express both a direct and indirect model integration and, thus, allows the definition of a hybrid model integration:

- The asserted or inferred types of the bound OwlFrame, or both. The value of the attribute is a set representing the OWL classes. Notice that the bound types of the OwlFrame are usually not exposed because they are seen as a technical feature of Mooop which is used behind the scenes.
- The asserted or inferred properties of the bound OwlFrame, or both. The value of the attribute is a set of tuples each representing an OWL property and its values. This can be seen as an indirect integration of parts of the OWL model.
- The values of a specific property of the bound OwlFrame. The value of the attribute is a set representing the values of the OWL property. This can be seen as a direct integration of parts of the OWL model.

One interesting extension which has not been investigated in more detail is the additional definition of an order of indirectly represented properties. Another useful extension is to provide an automatic classification after each change.

Method Binding

Mooop allows the annotation of methods of hybrid classes. Thereby, the method call is delegated to a handler defined by the method binding. This allows arbitrary behaviour. Again, Mooop allows the definition of custom method bindings. However, it also already provides bindings for two purposes: on the one hand,

one method binding provides a means for annotating methods of the hybrid class to trigger the classification of the bound OwlFrame.

On the other hand, Mooop allows for an OWL model-based method dispatch. For that, a method of a hybrid class can be annotated to be overwritten by another method of that class if some predicate becomes true. Thereby, it is possible to define several overwriting methods per base method. In order to ensure that an overwriting method m' is always executable in the context of a call of the overwritten method m, the parameters of m' must be contravariant and the return type covariant concerning the signature of m. For instance, assume the Java classes A and B whereby B is a subclass of A. Then a method test with the signature A test(B a) can be overwritten by a method testOverwrite with the signature B testOverwrite(A a) because the parameter of test is more specific and the return type of test is more general.

The basic binding defines a predicate which checks whether the bound OwlFrame is typed as a member of a specific OWL class or not. However, at runtime the predicates for several overwriting methods can become true which leads to ambiguities which method to execute. This can be solved by either throwing an exception, define a resolution mechanism, e.g., a priority, or execute an arbitrary chosen method, which is used by the basic binding. The described method dispatch is simple but it would be interesting to investigate more complex predicates. Ernst et al. describe a generic concept for method dispatching, which would be interesting to integrate into Mooop [16].

Conversion

The information the OwlFrame contains is OWL specific (see Chapter 5.2.2): the types are OWL classes, the properties either OwlFrames or OWL literals, and the same individuals are OwlFrames. However, the hybrid objects contain application-specific information, e.g., they are not related to OwlFrames but to other hybrid objects. Therefore, the binding has to perform a conversion between the application independent model of the OwlFrame and the application specific Java model. This conversion is very domain specific. Hence, Mooop provides a basic implementation which can be customised.

The basic binding represents both OWL atomic classes and complex classes as Manchester syntax expressions [26] in Java Strings. The conversion forth and back is, thus, pretty simple. The same applies for the conversion of the names of OWL individuals. OWL data types are converted to basic Java classes matching best, e.g., an XSD integer is converted into a Java Integer.

In order to convert an OwlFrame into a Java model object, a hybrid object has to be created which is bound to the OwlFrame. But a problem can arise if an OwlFrame o should be assigned to the object of a hybrid class hc which has the bound type $t(hc)$ but o has not the (asserted or inferred) type $t(hc)$. In this case an instantiation and binding of $[hc \rightarrow o]$ would change the OWL model since o gets a new asserted type. There are two approaches to handle this problem: (1) instantiate and bind $[hc \rightarrow o]$ anyway, or (2) omit o as a value and do not return it. On the one hand, Approach 1 asserts that the missing type $t(hc)$ of o was an error which is corrected by the binding, i.e., $i(o)$ should actually be of type $t(hc)$. On the other hand, Approach 2 sees the

type $t(hc)$ as a filter for the property values, i.e., the result should only contain OWL individuals of type $t(hc)$ and others should be ignored. Since the second approach is not an obvious semantics, Mooop uses Alternative 1 as the default.

The domain specific representation of an OwlFrame, i.e., a hybrid object, is in general not uniquely determined because the OwlFrame can have multiple types. For directly integrated properties the attribute mapping can convey the target hybrid class for the conversion. But this is not possible for indirectly integrated properties. Therefore, the OwlFrames in the values are converted to a special hybrid class which is bound to $\top$. Since $\top$ is the superclass of all classes, this instantiation is always possible and does not change the meaning of the OwlFrame.

5.3.2 Runtime Binding

The mapping and the structural binding together define the static integration of concepts and properties of an OWL model into an OO model. However, it is not yet determined how hybrid objects are integrated into the rest of the OO model at runtime and what reactions other objects, which interact with them, can expect. This is defined by the *runtime binding*. Notice that, in contrast to the structural binding, the runtime binding can not be customised. It determines the basic functionality of Mooop and, therefore, a customisation would be highly complex and error-prone.

Every hybrid object has a life-cycle which is determined by the Mooop framework. In order to allow the developers to control the life-cycle, a special component, the *Mooop manager*, offers an interface which allows to influence the life of hybrid objects. The functions of the Mooop manager and their results are:

create(hc) returns a hybrid object $[hc \rightarrow o]$ whereby $i(o)$ was newly created using the mapping function create(o, A) and A is the name of the hybrid class hc which was converted to a URI. hc is referred to as the *instantiation type* of $[hc \rightarrow o]$. This process comprises the binding of the hybrid object to an OwlFrame and, hence, the bound OWL classes of hc are added to the bound classes of o as described in Chapter 5.3.1.

load(n, hc) returns a hybrid object $[hc \rightarrow o]$ whereby $i(o)$ was loaded using the mapping function load(o, A) and A is the string n after the conversion to a URI. hc is the *instantiation type* of $[hc \rightarrow o]$. The rest of this process is similar to the process of create(hc).

deinstantiate$([hc \rightarrow o])$ de-instantiates $[hc \rightarrow o]$, i.e., the removal of the bound OWL types of the hybrid class hc from the OwlFrame o (see Chapter 5.3.1) and deletion of the association between $[hc \rightarrow o]$ and o, thus, invalidating $[hc \rightarrow o]$. After this, every call of a bound method of $[hc \rightarrow o]$ will cause an exception.

reinstantiate$([hc \rightarrow o], hc_1)$ re-instantiates o with hc_1, i.e., deinstantiation of $[hc \rightarrow o]$ and creation a new hybrid object $[hc_1 \rightarrow o]$ using load(n, hc_1) whereby n is the URI of $i(o)$ after conversion to a string. This can be seen as a type casting for hybrid objects.

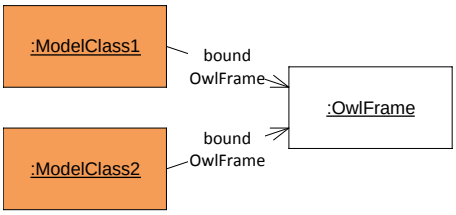


Figure 5.11: Several hybrid objects can be bound to the same OwlFrame object at runtime.

HoN : HC[OiN : OiT] → HoN : HC[OiN : OiT]	
HoN	Name of the hybrid object
HC	Instantiation type of the hybrid object (Hybrid class)
OiN	Name of the assigned OwlFrame
OiT	Comma separated asserted and inferred types of the assigned OwlFrame
→	State transition; the object before → turn into the object behind it

Table 5.3: Notation for describing hybrid objects.

delete($[hc \rightarrow o]$) removes $[hc \rightarrow o]$, i.e., de-instantiation of $[hc \rightarrow o]$ and all other hybrid objects $[hc_x \rightarrow o]$ which are bound to o, and invalidating of o by removing $i(o)$ using the mapping function delete(o).

isBindable($[hc \rightarrow o], hc_1$) returns true if $\text{getTypes}_{\text{a}}(o)$ contains $t(hc_1)$ (and bound types of subclasses of hc_1); otherwise false. This can be seen as a check whether $[hc \rightarrow o]$ can be cast to hc_1 without changing o.

The reaction of an object to a received message depends on its current state and implemented behaviour. Thereby, the state of hybrid objects is divided into a part modelled in OWL and a part modelled in OO. The integration of the OWL part of the state is determined by the mapping and structural binding. Thereby, it is important to note that several hybrid objects can share the OWL part of the state since several hybrid objects can be bound to one OwlFrame object at runtime as depicted in Figure 5.11.However, the integration of the OO part of the state and the behaviour is solely defined by the runtime binding. Generally, there are two alternatives for this: one is driven by the OO modelling formalism and the other is driven by the OWL modelling formalism.

In the following discussion of the different integration concepts, the scenario shown in Figure 5.12 is used to exemplify the different reactions upon messages. It consists of a hybrid class Vehicle and its hybrid subclasses Car and Ship. Each of these classes is bound to an OWL class of the same name, and these OWL classes are in a similar subclass hierarchy in the OWL model. Additionally, the short notation shown in Table 5.3 will be used for denoting hybrid objects. As an example, a:Vehicle[t:Car] denotes a hybrid object a which is an instance of the hybrid class Vehicle. Furthermore, a is bound to the OwlFrame t which has the assigned or inferred type Car.

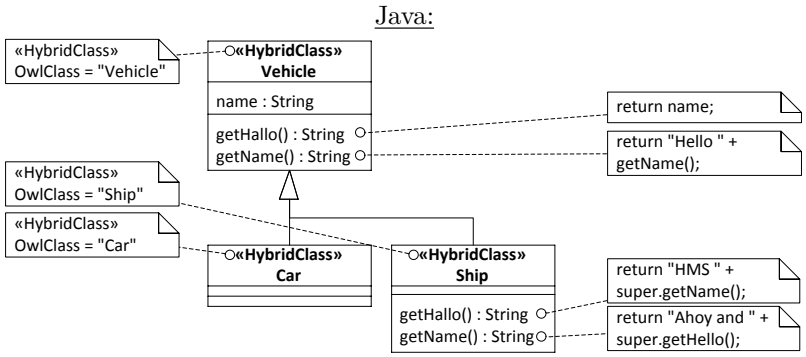


Figure 5.12: Java and OWL models for the vehicle example.

Java State

The Java state of a hybrid object comprises the values of all its pure Java attributes. These are all attributes of its hybrid class and its hybrid class's base types which are not bound by the structural binding. The runtime binding defines whether this state, or parts of it, should be shared between several hybrid objects. The following two questions are to be answered (see Figure 5.12):

1. Is the OO state shared between hybrid objects which are of the same type and assigned to the same OwlFrame?
   Example: a:Car[t:Car].name = b:Car[t:Car].name – If two hybrid objects a and b which are both instances of the hybrid class Car are assigned to the same OwlFrame t, is the state of a and b the same?

2. Is the OO state (or parts of it) shared between hybrid objects which are assigned to the same OwlFrame and which instantiation types are in a subclass relationship?
   Example: a:Car[t:Car].name = b:Vehicle[t:Car].name – If two hybrid objects a which is an instance of the hybrid class Car and b which is an instance of the hybrid class Vehicle are assigned to the same OwlFrame t and Car is a subclass of Vehicle, is the common part of the state of a and b the same?

Next, we discuss these two questions from both an OO-driven and an OWL-driven perspective.

An OO-driven approach keeps the state of a hybrid object encapsulated and separated from the states of other objects. However, it only allows for one hybrid object instance per combination of hybrid class and OwlFrame. Otherwise, a Java state would be almost useless at all. Imagine an OwlFrame t:Car is the

value of two properties u and r of another OwlFrame s. If a developer would instantiate t:Car via u with the hybrid class Car and via r with the same hybrid class Car he would get two separate hybrid objects a:Car[t:Car] and b:Car[t:Car] with separate states. The Question 1 is answered with with *yes*, the OO state is shared, and a and b are identical. However, Question 2 is answered with *no*, the state is not shared. The only part of the state of a hybrid object which is shared is the OWL state. The need for a shared Java state between different hybrid objects appears to be some kind of code smell [17], because this information should actually be modelled in OWL.

An OWL-driven approach shares the state between hybrid objects that are bound to the same OwlFrame. This is quite complex in Java. One solution could be the central storage of the OO state in the OwlFrame and the synchronisation of the local state of the hybrid objects with this before and after each method call. Another appealing solution is to create a single hybrid object of a hybrid class – Owl Frame combination on each inheritance level. Hence, the instantiation of a:Car[t:Car] would actually result in the creation of two hybrid objects a:Car[t:Car] and b:Vehicle[t:Car]. The hybrid object of a subclass (a:Car[t:Car]) would then delegate calls to methods which are not implemented in the direct type (Car) of the hybrid object to the hybrid object of the super class (b:Vehicle[t:Car]). Unfortunately this elegant solution can not be used because delegation can not imitate inheritance in Java [34]. Besides the implementation difficulties, the sharing of a state breaks the encapsulation principle of OOP languages which can be quite odd for developers: in each method of a class, it has to be ensured that the current state of the hybrid object is legal and has not been illegally manipulated by another object. As an example, imagine a:Vehicle[t:Car] and b:Car[t:Car] which share the value of the attribute name. a can manipulate name in a way which is not valid for b. However, this problem can be avoided since, once the states are somehow extracted from the hybrid objects to the OwlFrame, it can be controlled which of these states are actually merged. Thus, it can be adjusted whether the attribute values are unrestrictedly shared between hybrid classes, are shared between hybrid classes which are in a subclass relation, or only within one hybrid class, i.e., no sharing. That means that the answers to both Questions 1 and 2 depend on the sharing policy; however, in general they are *yes*.

Having carefully weighed the advantages and disadvantages of both approaches, it seems clear that the OO-driven state integration concept should be preferred. Despite its ability to flexibly control the degree of state sharing, the OWL-driven approach is disproportionately more complex. Furthermore, the need for a shared state is a sign of a non-optimal division of information between the OO model and the OWL model which should not be supported. Nevertheless, if it is necessary to share the OO state between hybrid classes, there are alternative OOP approaches for achieving this.

Java Behaviour

The behaviour of a hybrid object is solely defined within the Java model since there are no means for expressing behaviour in the OWL model. The behaviour of pure Java objects is defined by their type (or possible super types). Since Java

is a single typed OOPL, the behaviour is unambiguously determined. However, from a conceptual point of view, a hybrid object has several types, namely a single static instantiation type in Java and several dynamic, asserted and inferred OWL types. The runtime binding clearly defines the behaviour of a hybrid object. More precisely, it defines the behaviour of which hybrid class is executed by a hybrid object. The following questions are to be answered (see Figure 5.12):

1. What is the behaviour of a hybrid object which was instantiated with a superclass of another hybrid class, which is also applicable for this OwlFrame?
   Example: a:Vehicle[t:Ship].getName() – What is the result of the getName() method of a hybrid object a which is an instances of the hybrid class Vehicle and which is assigned to the OwlFrame t which has the type Ship?

2. What is the behaviour of a hybrid object which was instantiated with a super class of several other applicable hybrid classes?
   Example: a:Vehicle[t:Car,Ship].getName() – What is the result of the getName() method of a hybrid object a which is an instances of the hybrid class Vehicle and which is assigned to the OwlFrame t which has the types Ship and Car?

3. Are abstract hybrid classes allowed and can they be used to instantiate hybrid objects?

4. What is the behaviour of a hybrid object after the types of the assigned OwlFrame changed?
   Example: a:Vehicle[t:Ship].getName() → a:Vehicle[t:Vehicle].getName() – What are the results of the getName() method of a hybrid object a which is an instances of the hybrid class Vehicle and which is assigned to the OwlFrame t before and after t changes its type from Ship to Vehicle?

Next, we discuss these two questions from both an OO-driven and an OWL-driven perspective.

An OO-driven approach answers these questions on the basis of the instantiation type. The actual type of a hybrid object is the hybrid class which it was instantiated with and, therefore, the instantiation class defines the complete behaviour of a hybrid object. So, concerning Question 1 and 2 the result is in both cases the *name without "HMS"*. However, the answer to Question 3 is not so clear. On the one hand, in Java, it is not allowed to instantiate abstract classes. On the other hand, in a hybrid model, it can be useful to instantiate an abstract hybrid class with an OwlFrame, pass it around and later re-instantiate it with a concrete hybrid class. Imagine a method which returns Vehicles, whereby Vehicle is an abstract class in this case, and later each of these is checked whether it is a Car or a Ship and accordingly re-instantiated. If an abstract method of an abstract hybrid class is called, an exception is thrown. So, the answer to Question 3 should be *yes*. The problem of Question 4 is no issue at all since the Java type of the hybrid class remains the same and so its behaviour.

Despite its attractive simplicity, this approach has an immense shortcoming: the hybrid model is hard to extend because of the lost polymorphism for

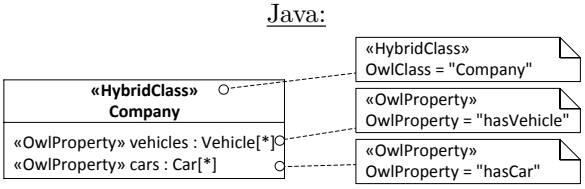


Figure 5.13: Java and OWL models for the extended vehicle example.

the OWL properties of hybrid objects. Imaging the scenario depicted in Figure 5.13 and a hybrid object a:Company[x:Company] which has one Car named t. In this case, the calls of getVehicles() and getCars() yield different hybrid objects b:Vehicle[t:Car] and c:Car[t:Car] with diverse behaviour and state (see Chapter 5.3.2). As a consequence, the developer would have to check and reinstantiate every result value. This is even worse if the hybrid model is later extended by new hybrid subclasses of existing hybrid classes. In this case all of these check routines have to be changed.

One can also think of modifying the instantiation of the former concept to allow only the instantiation of the most specific types. So, concerning Question 1 the developer would get an exception because he has to instantiate a Ship. But the drawbacks of this concept become apparent when looking at Question 2 since, in OWL, it is not guaranteed that an OWL individual has only one most specific type. This modification is also unsuited for evolving Java models. A new hybrid subclass of an existing hybrid class will cause lots of exceptions in the program since this subclass is never used in the old code.

In contrast to the OO-driven approach, an OWL-driven approach defines the behaviour of a hybrid object based on the OWL types of the assigned OwlFrame. Thus, the instantiation type of a hybrid object does not determine a single *actual type* which defines the behaviour of a hybrid object but a set of possible types. These possible types are the instantiation type itself and all its subclasses. The actual type of a hybrid object is then the most specific type out of this set which is applicable for the assigned OwlFrame. Thus, for Question 1 the actual type of a is Ship and the returned *name starts with "HMS"*. However, a problem already arises for Question 2 because the most specific type is not uniquely determined. So, some resolution mechanism has to determine the actual type. A combined priority-exception based approach, which chooses one class based on a priority or, if no priority is given, throws an exception, seems to be very suitable. Hence, the actual type of a in Question 2 can be either Car, Ship, or an exception is thrown. The problem of Question 3 only occurs if the most specific hybrid class is an abstract class. The possible solutions are the same as in the OO-driven approach: exception upon instantiation or exception upon abstract method call. However, in an OWL-driven approach the latter is definitely more sensible since the most specific type can change during execution and the hybrid object can

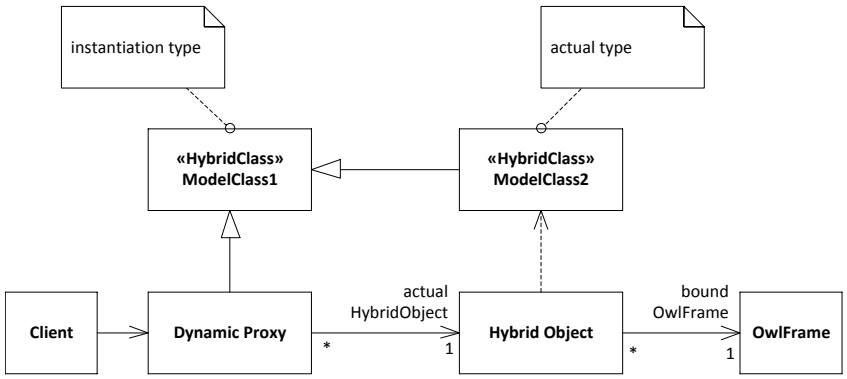


Figure 5.14: State pattern-based design of the OWL-driven runtime binding.

change its actual type in parallel. This already gives an answer for Question 4: the actual type of a hybrid object is dynamic and so its behaviour.

It is quite obvious that the implementation of such a concept is quite complex in a statically typed OOPL like Java. A first idea is to make use of the *strategy pattern* [18]. However, this implies that the developers have to write their methods as strategies, which is quite awkward, odd and indirect. Another, more sophisticated solution is a combination of *state pattern* and *proxy pattern* [18] depicted in Figure 5.14. The developer interacts with a dynamic proxy which is a subclass of the instantiation type of the instantiated hybrid object. Behind the scenes, the proxy delegates the method calls to a hybrid object, which is an instance of the actual type. When the actual type has to change, the state from the old actual hybrid object is extracted and copied to a new hybrid object which, subsequently, replaces the old actual hybrid object. However, in this way, the methods of the new type are working with a possibly illegal OO state from the former type. The problem is similar to the shared state problem discussed in Chapter 5.3.2. Fortunately, it can be mitigated by introducing a special method or constructor in the hybrid classes which checks and, as the case may be, corrects the old OO state. Another, not obvious issue arises when a hybrid object changes its type to a super type of the current type and back again. Since there is no general shared state (see Chapter 5.3.2), the part of the state, which is defined by the subtype, is lost. However, this can be an intended course of action because the hybrid object actually changed its type. Finally, notice that in this approach the instanceof operation does not check against the hybrid class for the behaviour of the hybrid object but against the instantiation type. This is reasonable because in this way normal Java type casts are allowed. However, it reveals the necessity for a new special hybrid class instance check operation.

In order to decide for the OO-driven approach or the OWL-driven approach, both have to be evaluated in context of the whole Mooop framework. However, it seems more suited to compare the OWL-driven object behaviour integration approach with a combination of the OO-driven object behaviour integration approach and the dynamic method dispatch shown in Chapter 5.3.1. The latter is as powerful as the former; however, they differ in their implications on modelling. The extended OO-driven approach separates the dispatch of the

pure OO methods, which are handled according to the instantiation type of the hybrid object within the OO hierarchy, and the bound methods, which are executed according to the OWL types of the assigned OwlFrame. Hence, the two class hierarchies can evolve separately and are only loosely coupled. This approach will lead to a flat class hierarchy of hybrid classes because there is no need for subclasses since the specialised methods can be also implemented in the base class. Even more, in the light of the earlier mentioned issue of the recurring re-instantiation of hybrid objects with more specialised classes, the usage of subclasses seems unsuited. Therefore, it is foreseeable that the use of this approach will create extensive hybrid classes which implement both general cases and specialisations. This breaks the Single Responsibility Principle of OO design and should be avoided because it makes classes harder to understand and change.

The OWL-driven approach can be seen as a dynamic method dispatch. Instead of checking before each call of a bound method to which specific one it should be dispatched, this decision is done once at classification for bound and pure OO methods. However, in both cases, the problem of not uniquely determined methods or classes, respectively, is apparent. The hybrid classes in this concept are presumably easier to read since they encapsulate only state and behaviour for the bound OWL class. Additionally, the model in this concept is easy to extend by just adding new hybrid subclasses whereas in the OO-driven model, old classes have to be changed. On the other hand, this can lead to a hybrid class inheritance hierarchy which imitates the respective OWL class hierarchy since the developers will write small hybrid classes containing specialised behaviour for specialised OWL classes. This is an unintended modelling implication since the idea of hybrid modelling is to only represent a fraction of the numerous OWL classes within the OO model. This reveals another implication on the modelling: the hierarchies of the hybrid classes and the OWL classes in OWL are tightly coupled.

Having carefully weighed the advantages and disadvantages of both approaches, we have chosen the OO-driven behaviour integration concept in combination with the dynamic method dispatch for Mooop. It offers the best support for the idea of hybrid modelling, i.e., to represent only a few top level OWL concepts as hybrid classes. Hence, it is supposed to be easier to understand.

5.4 General Approach for Hybrid Integration

Although the extensive former chapters probably gave a different impression, the basic concept of Mooop is surprisingly simple: an adaptive object model represents the complex knowledge of an OWL model in an indirect manner, a mapping performs the translation between the ontology and the AOM, and a binding enables a hybrid access from the Java model to the AOM through hybrid classes. This overview reveals that the Mooop idea can be generalised into a concept for a generic hybrid integration of any structural modelling approach like Topic Maps [31] or XML into an OOPL like Java.

We think the generalisation of Mooop is a promising project. However, there are some complex issues to be solved. First, the OwlFrame has to be generalised

to suit other modelling languages than OWL. We think that the division into asserted and implied information is generally interesting but probably the OwlFrame has to be extended to represent further modelling elements. Second, the mapping already provides an abstraction layer from the modelling language. However, its interface has to be extended in accordance to the OwlFrame in order to provide a reasonable access to models of other languages. Third, the binding has to be independent from the modelling language at all. Therefore, it requires only little adaptation. However, the generalisation of Mooop is still subject to further research and, thus, not presented in detail in this work.

Chapter 6

Design of Mooop Prototype

We implemented most parts of the Mooop concept in a prototype in order to concrete it, show its feasibility, evaluate it, and facilitate further research. This chapter explains the architecture of the prototype and describes some interesting design decisions. For a usage example of the prototype please refer to Chapter 7.1.1.

6.1 Architecture

A main requirement for a generic integration is that the integration must be adaptable to the specific modelling guidelines in a project. The Mooop concept reflects this fact by introducing the mapping and binding. Accordingly, the architecture of the prototype separates the mapping and binding into components and, thus, allows the extensibility of both.

Figure 6.1 shows the architecture of an application using the Mooop prototype. Mooop is based on the *OWL API* for accessing the OWL model. As shown in Chapter 3.2, the OWL API is a framework for the indirect integration of an OWL model into Java and, hence, offers a flexible access to the ontological knowledge. We allow this hard dependency for our prototype but suggest to abstract this in a final implementation in order to allow the usage of different frameworks. All classes of the OwlFrame's AOM are contained in the *OwlFrame* component. The OwlFrame accesses the OWL model through the *Mappings* component which contains the predefined mappings of Mooop. Furthermore, the OwlFrame component is accessed by the *Bindings* component which contains the predefined bindings of Mooop. The domain model of the application usually consists of both hybrid classes, which are defined using the Bindings component, and pure Java classes. However, in order to enable the adaptation of the integration to the project needs, the application can also define a custom mapping and binding. The prototype can be configured to use them instead of the predefined ones. In order to facilitate reuse, the custom mapping and binding components can access the predefined ones in order to reuse their functionality. At runtime, the access to the Mooop framework for, e.g., instantiating new hybrid objects is performed through a façade called *MooopManager*.

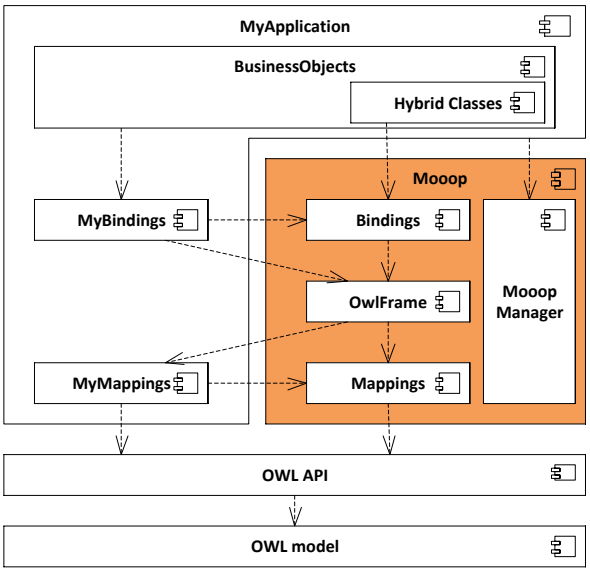


Figure 6.1: Architectural view on the Mooop framework and an application using it.

6.2 Implementation

The design of the prototype is in general aligned with the concept. For instance, the OwlFrame is a Java class with the structure and behaviour described in Chapter 5.1 (see the Java package uk.ac.manchester.cs.mooop.owlFrame). So, the following description of interesting parts of the design concentrates on interesting design decisions which have not been mentioned before and refers to the Chapter 5 where appropriate. For more details on the implementation, please refer to Appendix D.

6.2.1 Mapping

The mapping defines the integration of the OWL model into the generic OwlFrame. More precisely, the OwlFrame calls methods of the mapping through the Java interface Mapping which, vice versa, modify the OwlFrame. The interface extends 4 other interfaces, TypeMapping, PropertyMapping, IndividualMapping, and ReasoningMapping, which offer the methods introduced in Chapter 5.2. Please refer to Figure C.1 in Appendix C for a detailed view on them. The Mapping interface defines a contract which each predefined and custom mapping has to fulfil. In order to do this, the mapping can access the OWL ontology and OWL reasoner through the OWL API.

The mapping can be seen as a stateless *strategy* [18] shared by all OwlFrames. Hence, the mapping should not have a state which is modified by methods defined in Mapping. On the contrary, the methods are supposed to manipulate a MappingContext which is passed as a parameter. This MappingContext allows the access to the state of the OwlFrame which called the mapping, the OWL

ontology which contains the OWL model, and the OWL reasoner for the OWL ontology. In other words, the MappingContext contains all information a Mapping method needs to perform its behaviour. An example for a mapping method can be seen in Listing 6.1. This source code performs exactly the modifications defined for the method $\text{addType}_\text{a}(o,t)$ in Chapter 5.2.1. In the prototype, the mapping is also responsible to establish a lazy loading and caching of data from the OWL model to improve performance. However, this can be probably improved by some more generic concept.

```
1  public class OwlIndividualMapping implements Mapping {
...
3    @Override
     public void addAssertedType(final MappingContext o, final
         OWLClassExpression t) {
5      // add i(o) in t to ontology
       final OWLNamedIndividual i = o.getIndividual();
7      OWLClassAssertionAxiom axiom = configuration.getOwlDataFactory().
           getOWLClassAssertionAxiom(t, i);
       configuration.getOwlOntologyManager().addAxiom(configuration.
           getMainOntology(), axiom);
9      // add t to asserted types of o
       o.getAssertedTypes().add(t);
11   }
...
13 }
```

Listing 6.1: Example implementation of $\text{addType}_\text{a}(o,t)$ from the type mapping.

The use of the strategy pattern for the mapping is an important design decision. An alternative would be to merge the mapping with the OwlFrame and, thus, let the OwlFrame directly access the OWL ontology. In this case, a mapping could be reused by creating a subclass of it, or make use of the *decorator pattern* [18]. However, in both cases it is only possible to reuse one mapping. If one would like to extend two mappings, he has to ensure that the state of both mappings is in synch. On the contrary, our strategy approach allows the flexible combination of an arbitrary number of mappings for defining a custom mapping. This is because the methods of the mapping are only changing the state of the parameters but not a state of the mapping.

The former description provided just an overview of the mapping of Mooop. Please refer to the Java package uk.ac.manchester.cs.mooop.mapping of the prototype source code for further details (see Appendix D).

6.2.2 Binding

The implementation of the binding facility is inspired by the JPA. Accordingly, the binding uses Java annotations for a declarative definition of hybrid classes. However, in contrast to the JPA the binding has to be customizable to the needs of a project. In other words, it has to be possible to define new annotations with a new meaning.

As shown in Chapter 5.3.1, there are three types of binding: class binding, attribute binding, and method binding. However, in order to keep it simple we are only distinguishing two different types of annotations in hybrid classes: class annotations which are declared for the class, and method annotations which are

declared for the methods. Hence, it is not possible to bind the attributes of a hybrid class but instead the getter and setter methods for the attribute.

The binding facility of Mooop is actually a framework for defining annotations and assigning annotation handlers. The predefined binding annotations in the prototype are declared in the same way a custom binding annotation would be defined using the framework. In order to create a new binding annotation, a Java annotation has to be annotated with a *meta-annotation*. There are two of these meta-annotations defined by the binding facility: a class and a method meta-annotation. Both have one mandatory element which defines the handler for marked annotation. In the following, we present these meta-annotations and their usage in more detail.

In the prototype the class binding is performed by annotating a class with the annotation @HybridClass (see Chapter 5.3.1). The Listing 6.3 shows the source code of the annotation defining the mandatory value which represents the OWL class this annotated hybrid class should be bound to. However, more important is the meta-annotation @MooopClass depicted in Listing 6.2: it defines that the annotation @HybridClass is a class binding annotation for Mooop. Furthermore, it determines a handler which is responsible for the hybrid class and which has to implement the Java interface MooopClassHandler. As shown in Listing 6.2, a MooopClassHandler instance has to determine the bound classes of the hybrid class (defined in BoundTypeProvider) and can initialize an OwlFrame which is bound to a hybrid object of the hybrid class. Notice, that the BoundTypeProvider interface does not restrict that a hybrid class can only be bound to one OWL class. During the instantiation of a hybrid class, the Mooop framework binds the MooopClassHandler to the OwlFrame and, thus, adds the provides bound types to the bound types of the OwlFrame object. The HybridClassHandler provided by the prototype is rather simple as Listing 6.3 shows. For an example of the usage of the standard class binding, refer to Listing 7.1. One can easily define a custom class binding by defining an own Java annotation which itself is annotated with @MooopClass, and write an own handler for it.

The annotations for methods are defined in the same way as the class annotations. Listing 6.5 shows the declaration of the predefined OwlProperty annotation. The meta-annotation MooopMethod has an element which defines the handler for this annotation. In case of the method binding annotations the handler has to implement MooopMethodHandler as shown in Listing 6.4. The method initialize(...) is called upon instantiation of a new hybrid object and is supposed to initialize the handler. If an annotated method is called, the call is intercepted by the Mooop framework and instead the call(...) method of the method handler executed. The parameters of the method represent the hybrid object for which it was called[1] and the arguments of the original method call. An interesting parameter is the MethodInvocator originalMethod: it is an object which represents the continuation of the original method, i.e., it can be used to execute the annotated method. However, the OwlPropertyHandler never calls the annotated method. The return value of the call(...) method is then returned to the called of the original method. Listing C.1 in Appendix C shows a part of the OwlPropertyHandler in order to exemplify this concept. Besides the OwlProperty annotation, the OwlType annotation is the most widely used.

[1] Notice that this hybrid object is an instance of the hybrid class passed with initialize.

```
1  public @interface MooopClass {
     /** Handler managing the hybrid class. */
3    Class<? extends MooopClassHandler> value();
   }
5
   ---------------------------------------------------------------
7  public interface BoundTypeProvider {
     /** @return the bound types of this Provider */
9    Set<OWLClassExpression> getBoundTypes();
   }
11
   ---------------------------------------------------------------
13 public interface MooopClassHandler extends BoundTypeProvider {
     /** Initialize the handler. */
15   void initialize(final Class<?> hybridClass, MooopConfiguration
         mooopConfiguration)
         throws MooopConfigurationException;
17   /** Initialize the OwlFrame after the binding was performed. */
     void initializeOwlFrame(OwlFrame owlFrame) throws
         MooopConfigurationException;
19 }
```

Listing 6.2: At the top the MooopClass meta-annotation for defining a class binding, in the middle the BoundTypeProvider interface which is extended by the MooopClassHandler shown below.

```
1  @MooopClass(HybridClassHandler.class)
   public @interface HybridClass {
3    /** The name of the bound OWL class. */
     String value();
5  }
7
   public class HybridClassHandler implements MooopClassHandler {
9    ...
     public Set<OWLClassExpression> getBoundTypes() {
11     return boundTypes;
     }
13   public void initialize(final Class<?> hybridClass, final
         MooopConfiguration mooopConfiguration)
         throws MooopConfigurationException {
15     HybridClass annotation = hybridClass.getAnnotation(HybridClass.class
           );
       String owlType = annotation.value();
17     try {
         OWLClassExpression boundType = mooopConfiguration.
             getDomainFrameConverter().convertOwlClassExpression(owlType);
19       boundTypes.add(boundType);
       } catch (MooopParserException e) {
21       throw new MooopConfigurationException(e);
       }
23   }
     ...
25 }
```

Listing 6.3: At the top the annotation HybridClass from the prototype and below the implementation of the defined handler for teh annotation HybridClass.

It allows to access the types of an OwlFrame. Please refer to Listing C.2 in Appendix C for more details about this method annotation.

```
public @interface MooopMethod {
  /** Handler called upon a call of the annotated method */
  Class<? extends MooopMethodHandler> value();
}

---

public interface MooopMethodHandler {
  /** This class represents the original method of the hybrid object. */
  public static interface MethodInvocator {
    /** execute the method with the original parameters */
    Object run() throws Throwable;
    /** execute the method with the given parameters */
    Object run(Object[] arguments) throws Throwable;
  }
  /** This method is called whenever the annotated method of the hybrid
      object is called. */
  Object call(Object hybridObject, Object[] arguments, MethodInvocator
      originalMethod);
  /** This method is called upon initialization of the hybrid object. */
  void initialize(final Class<?> hybridClass, final Method
      annotatedMethod, final OwlFrame assignedOwlFrame,
      MooopConfiguration mooopConfiguration) throws
      MooopConfigurationException;
}
```

Listing 6.4: At the top the MooopMethod meta-annotation and below the MooopMethodHandler interface necessary to implement method bindings.

```
@MooopMethod(OwlPropertyHandler.class)
public @interface OwlProperty {
  /** type of properties which are returned */
  AxiomType axiomType() default AxiomType.ASSERTED_OR_INFERRED;
  /** type of method */
  MethodType methodType() default MethodType.AUTO;
  /** property name which should be directly integrated. "" means
      indirect integration of all properties */
  String value() default "";
  /** type of return value */
  Class<?> valueType() default Thing.class;
}
```

Listing 6.5: The OwlProperty annotation from the prototype.

The two designated bindings for methods introduced in Chapter 5.3.1 are implemented like the bindings for attribute shown above. The simple annotation for designating a method to trigger the classification of an OwlFrame is shown in Listing 6.6. In contrast to this, the annotation for the OWL model-based method dispatch, OwlDispatch, is more complicated as can be seen in Listing 6.6: The value of the annotation allows to define an array of OwlDispatchEntry each representing a combination of OWL class and method. The dispatch implemented by OwlDispatchHandler works as described in Chapter 5.3.1. For an example of the usage of the dispatching facility, refer to Listing 7.4.

Besides the definition of the hybrid classes, the binding also comprises a conversion facility to translate objects from the OWL-centric OwlFrame component into objects of the domain-specific business objects component and vice versa. The binding facility of the prototype defines two interfaces, one for each translation direction, which have to be implemented by a custom as well as by

```
@MooopMethod(ClassifyHandler.class)
public @interface Classify {
}
```

```
@MooopMethod(OwlDispatchHandler.class)
public @interface OwlDispatch {
  /** Dispatch entry defining the method which the call should be
      dispatched if the bound OwlFrame is of type owlType. */
  public static @interface OwlDispatchEntry {
    /** Name of method the call should be dispatched to. */
    String methodName();
    /** Type of the OwlFrame (OWL class) which enabled dispatch.  */
    String owlType();
  }
  /** Array of entries defining the dispatch. */
  OwlDispatchEntry[] value();
}
```

Listing 6.6: At the top the simple Classify annotation an below the OwlDispatch annotation used to define an OWL model-based method dispatch.

the predefined converters. Listing 6.7 shows the interfaces for the translation of domain model data to OwlFrame data (DomainFrameConverter) and the other way around (FrameDomainConverter). The above mentioned handler for hybrid classes and their methods can access these converters for their task.

```
public interface DomainFrameConverter extends ConfigurationEntry {
  IRI convertIri(Object name);
  OWLClassExpression convertOwlClassExpression(Object typeName) throws
      MooopParserException;
  OWLLiteral convertOwlDataPropertyValue(Object values) throws
      MooopParserException;
  OwlFrame convertOwlObjectPropertyValue(Object values) throws
      MooopParserException;
  OWLPropertyExpression<?, ?> convertOwlPropertyExpression(Object
      propertyName) throws MooopParserException;
}
```

```
public interface FrameDomainConverter extends ConfigurationEntry {
  <T> T convertIri(IRI iri, Class<T> returnType);
  <T> T convertOwlClassExpression(OWLClassExpression classExpression,
      Class<T> returnType);
  <T> T convertOwlPropertyExpression(OWLPropertyExpression<?, ?>
      propertyExpression, Class<T> returnType);
  <T> T convertOwlPropertyValue(OwlFrame propertyValues, Class<T>
      returnType);
  <T> T convertOwlPropertyValue(OWLLiteral propertyValues, Class<T>
      returnType);
}
```

Listing 6.7: At the top the DomainFrameConverter interface for the translation of domain model data to OwlFrame data, and below the FrameDomainConverter interface for the other way around.

The binding facility of the Mooop prototype can be seen as a means to define a DSL for accessing the information of an OwlFrame in a domain-specific manner. Therefore, we can imagine far more complex binding annotations which extend the expressiveness of the binding. For instance it could be interesting to evaluate the idea of *annotation chains*, i.e., the nesting of annotations. The Listing 6.8 shows an example of an annotation chain: after executing the method

handler for @OwlProperty the method handler for @Classify is called. As a result, after adding the passed topping the OwlFrame is classified. However, it is not clear whether this is sensible and, thus, the bindings implemented in the prototype do not support this.

```
1  @Classify(@OwlProperty("hasTopping"))
   public void addTopping(final Topping parmaHamTopping) {
3    // will not be executed
   }
```

Listing 6.8: Vision of an annotation chain.

The former descriptions provided just an overview of the binding of Mooop. Please refer to the Java package uk.ac.manchester.cs.mooop.binding of the prototype source code for further details (see Appendix D).

6.2.3 Hybrid Objects

Hybrid objects are the instances of hybrid classes. They are bound to an OwlFrame and execute a MethodHandler upon the calls of annotated methods. Since the method delegation is not implemented in the hybrid class, it is not possible to instantiate a hybrid class normally. The behaviour of the hybrid classes has to be dynamically extended to include the delegation to the method handlers. Mooop uses the *Code Generation Library* (cglib) [7] to create dynamic proxies classes of hybrid classes, which are subclasses of the hybrid class, at runtime. The cglib allows the definition of MethodInterceptors to which a method call is delegated. The Mooop proxies are intercepting all annotated methods of a hybrid class and delegate them to the MethodHandler which is defined by the annotation.

Figure 6.2 depicts the simplified control flow for a call of an annotated method of a hybrid object. First, the method call to the cglib-Proxy, i.e., the hybrid object, is delegated to a MooopMethodInterceptor. This interceptor has been created during the creation of the proxy object. It knows the method handler for the called annotated method and forwards the call through the call(...) method. Eventually, the method handler (here OwlTypeHandler) accesses the bound OwlFrame to get or set ontological information (here addAssertedType(...)) and the OwlFrame, subsequently, calls a method from the mapping to perform the request (here addAssertedType(...)). Notice that a call to a non-annotated method is directly delegated to this method in the hybrid class.

This approach is an instance of the *proxy pattern* [18]. In contrast to the original pattern the cglib approach does not need to define the proxy classes at development time but dynamically creates them at runtime. However, just like in the pattern, the clients of the Mooop framework are never working with objects of the original hybrid class but with objects of the dynamic proxy.

The former description provided just an overview of the hybrid objects of Mooop. Please refer to the Java package uk.ac.manchester.cs.mooop.hybridObject of the prototype source code for further details (see Appendix D).

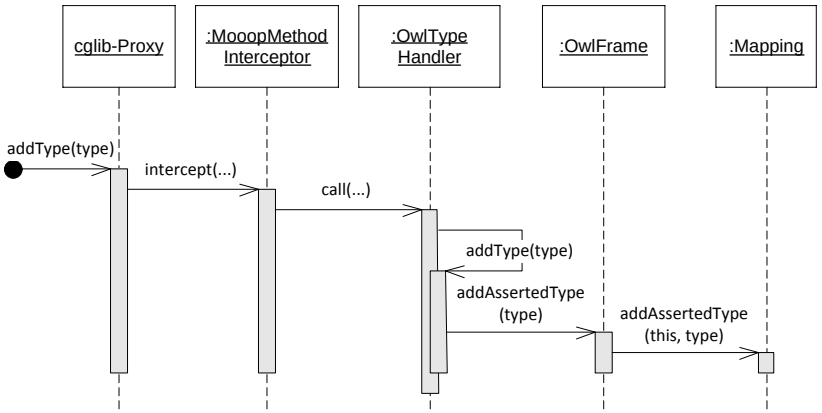


Figure 6.2: Simplified control flow for a method call.

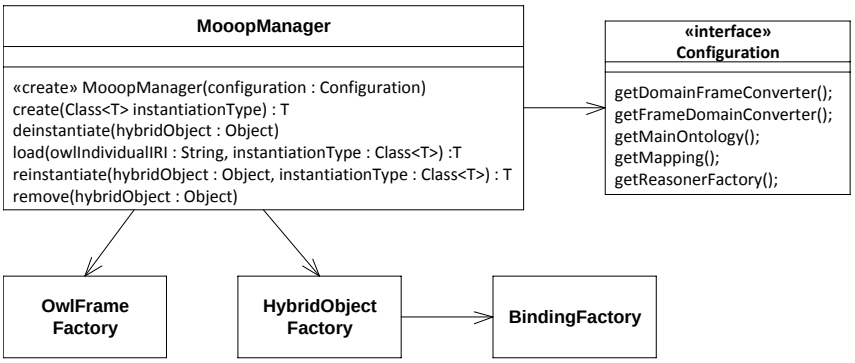


Figure 6.3: The MooopManager and important helper classes.

6.2.4 MooopManager

The MooopManager is the central façade for using the Mooop framework. It is the implementation of the Mooop manager introduced in Chapter 5.3.2 and provides the described interface. Figure 6.3 depicts the MooopManager and important related classes. In order to create the MooopManager, a data container called Configuration has to be provides by the developer: the configuration defines the mapping, converters, OWL ontology, and OWL reasoner which should be used. During the initialisation, the MooopManager loads the ontology and checks its consistency. Hence, changes on the OWL ontology from external programs are not visible to the MooopManager once it has been instantiated. To perform its behaviour, the MooopManager uses several factories which are part of the Mooop prototype: the OwlFrameFactory is responsible for creating and loading of OwlFrames and the HybridObjectFactory is responsible for creating hybrid objects, i.e., the proxy objects. So, HybridObjectFactory is also responsible to enforce that there is only one hybrid object per hybrid class – OwlFrame combination as mentioned in Chapter 5.3.2. In order to instantiate the bindings for a hybrid class, it uses the BindingFactory which creates and initialises the bindings.

The MooopManager is comparable to the EntityManager of the JPA. Similarly, the MooopManager defines important life cycle methods for the entities of the Mooop framework, that is the hybrid objects. They are implemented as described in Chapter 5.3.2. For instance, Figure 6.4 shows a simplified diagram of the instantiation of a hybrid class type. First, a new OwlFrame is created using the OwlFrameFactory which goes back to the mapping in order to create a new OWL individual. Using this new OwlFrame, the HybridObjectFactory can be used to create a new hybrid object. In order to analyse the hybrid class, the HybridClassAnalyser is used: first it analyses the Mooop class annotation and creates the class handler using the BindingFactory. This also binds the class handler to the OwlFrame. Afterwards, it gathers all Mooop related annotations and creates the method handlers using the BindingFactory again. Finally, the hybrid object is created and, eventually, returned to the original caller. This should give an idea of the interplay of the Mooop framework, the mapping, and the binding.

The former description provided just an overview of the MooopManager. Please refer to the Java package uk.ac.manchester.cs.mooop of the prototype source code for further details (see Appendix D).

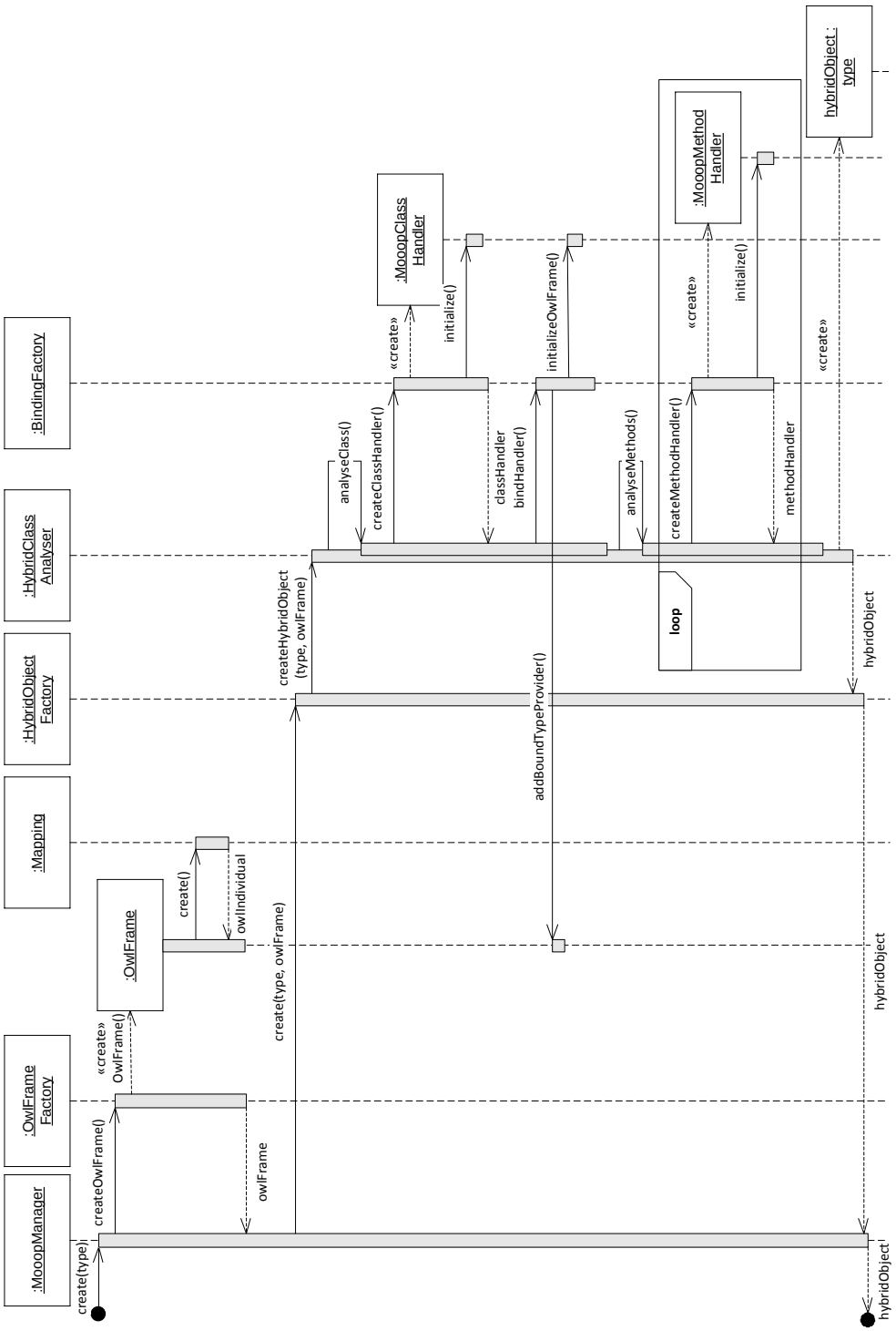


Figure 6.4: Simplified control flow of the create(...) method of the MooopManager which creates a new hybrid object.

Chapter 7

Evaluation of Mooop Concept

This chapter presents the results of an evaluation of the Mooop approach using the prototype described in Chapter 6. It shows the feasibility and usefulness of the concept on two implemented case studies, validates that Mooop fulfils the requirements for a generic integration, and outlines some ideas about a methodology for developing applications with Mooop.

7.1 Implementation of Case Studies

The two case studies described in Chapter 4.2 have been implemented to show the feasibility of the Mooop concept. We have developed the Java classes and OWL ontology for the domain model, and a custom mapping and binding if necessary. Since Mooop concentrates on the implementation of a domain model, we have not developed any *graphical user interface* (GUI) for the applications, but instead JUnit test cases performing interesting workflows with the models.

7.1.1 Pizza Configurator

The pizza configurator is our primary case study for evaluating the Mooop concept because it makes use of most features. We used the modified pizza ontology, developed the Java part of the domain model, and defined several workflows implementing common usage patterns. These patterns share the same general structure: it begins with the creation of an empty pizza which is, subsequently, topped with several toppings and, finally, information about the pizza, e.g., its price, are queried. The actual ordering of the configured pizza is omitted because this step is not related to an OWL-OO integration.

In the following, we outline how a developer would implement the pizza configurator. This can be seen as a scenario-based evaluation method where we focus on several common integration problems and show how they can be solved elegantly using the Mooop prototype. Although the discussion is not based on

the outcomes of experiments with real developers but on plausible arguments, we are still able to show the advantages of Mooop for the development of ontology-based applications. However, we do not evaluate a sophisticated methodology for hybrid modelling with Mooop. For a discussion of this issue, please refer to Chapter 7.3.

As with most hybrid models, the developer starts the implementation by picking several top level classes from the OWL model and represents them directly in Java. The top level classes should be classes which are often used, generic enough to represent lots of subclasses, and specific enough to define a distinct concept. In our case study these concepts are Pizza and PizzaTopping. Mooop allows for an declarative binding of Java classes to OWL concepts using Java annotations. This approach eases the declaration and keeps the integration code distinguishable and separated from the model code. For instance, Listing 7.1 shows the simplest version of the hybrid class Pizza which is bound to the OWL class Pizza using the predefined annotation @HybridClass.

```
@HybridClass("Pizza")
public class Pizza {
}
```

Listing 7.1: The hybrid class Pizza which is bound to the OWL class Pizza.

Now, the toppings and pizza have to be connected, so that a pizza can have several toppings. This is accomplished by directly integrating the OWL property hasTopping. So, the developer creates a getToppings(), a addTopping(), and a removeTopping() method in the Pizza class and annotates these methods with the Mooop annotation OwlProperty and, thus, binds them to a property from the OwlFrame. Listing 7.2 shows the result. The listing also shows a method for performing a classification (classify()), and a method to query all OWL types (asserted or inferred) of a hybrid object (getTypes()).

```
@HybridClass("Pizza")
public class Pizza {
  @OwlProperty("hasTopping")
  public void addTopping(final Topping topping) {
    // will not be executed
  }
  @Classify
  public void classify() {
    // will not be executed
  }
  @OwlProperty(value = "hasTopping", valueType = Topping.class)
  public Set<Topping> getToppings() {
    return null; // will not be executed
  }
  @OwlType(axiomType = AxiomType.ASSERTED_OR_INFERRED)
  public Set<String> getTypes() {
    return null; // will not be executed
  }
  @OwlProperty("hasTopping")
  public void removeTopping(final Topping topping) {
    // will not be executed
  }
}
```

Listing 7.2: The elaborated Pizza class which allows to handle toppings.

At this point, the model can already be used. Listing 7.3 contains a simple workflow showing the initialisation of the MooopManager with the ontology *./eks/pizza.owl*, the creation of a pizza and toppings and the query of the types of the pizza. Notice that mozzarella represents a newly created OWL individual, whereby, tomato represents an OWL individual loaded from the ontology. So far, this example works completely with a predefined binding and mapping. Therefore, the reasoning is based on an OWL individual classification.

```
1  @Test
   public void PizzaWolkflow1() throws MooopConfigurationException,
       MooopParserException {
3    // init MooopManager
     final DefaultConfiguration configuration = new DefaultConfiguration(
         new File("./eks/pizza.owl"));
5    mooopManager = new MooopManager(configuration);
     // pure pizza
7    pizza = mooopManager.create(Pizza.class);
     assertArrayEqualsIgnoringOrder(new String[] { "Thing", "Food", "
         DomainConcept", "Pizza" }, pizza.getTypes());
9    // create and add mozzarella topping
     Topping mozzarella = mooopManager.create(Topping.class);
11   mozzarella.addAssertedType("Mozzarella");
     pizza.addTopping(mozzarella);
13   // load and add tomato topping
     Topping tomato = mooopManager.load("tomato", Topping.class);
15   assertArrayEqualsIgnoringOrder(new String[] { "Tomato", "PizzaTopping"
         }, tomato.getAssertedTypes());
     pizza.addTopping(tomato);
17   // classify
     pizza.classify();
19   // check
     assertArrayEqualsIgnoringOrder(new Object[] { tomato, mozzarella },
         pizza.getToppings());
21   assertArrayEqualsIgnoringOrder(new String[] { "Thing", "Food", "
         DomainConcept", "Pizza" }, pizza.getTypes());
   }
```

Listing 7.3: Simple workflow creating a pizza Margherita.

Notice that in Line 21 in Listing 7.3 the system does not infer that pizza is a pizza Margherita. Having the definition of the pizzas from Chapter 4.2 in mind, this is not surprising because the OWL individual asserts the tomato and mozzarella topping but can still have other toppings as well. This is due to the OWA of OWL. However, if the OWL individual can have further toppings, it is not necessarily a pizza Margherita. Figure 7.1 shows a part of the ontology and an OWL individual which exemplifies the issue. Fortunately, this problem can be solved by adding an additional type to the OWL individual representing the pizza which states that it does not have any further toppings. For the example in Figure 7.1, this would mean $i_P \in \forall \mathsf{hasTopping}.\{i_M, i_T\}$. If i_P gets this additional type then it is inferred to be a Margherita. This is a good example for a custom integration logics. Therefore, the pizza configurator uses a custom mapping which adds a type to all OWL individuals which states that they have no more topping property assertions than the asserted ones.

The most interesting property from the OWL model is the hasPrice property. Every topping has an explicit price because every topping is a subclass of $\mathsf{hasPrice} : x$ whereby x is some integer value. Similarly, every defined pizza has an associated price. However, to get this x for a pizza, the custom mapping has to be extended even more: the reason for this is that the OWL API does not

$$
\begin{aligned}
\mathsf{Margherita} = & \mathsf{Pizza} \\
& \sqcap \exists \mathsf{hasTopping}.\mathsf{Mozzarella} \sqcap \exists \mathsf{hasTopping}.\mathsf{Tomato} \\
& \sqcap \forall \mathsf{hasTopping}.(\mathsf{Mozzarella} \sqcup \mathsf{Tomato}) \\
i_P \in & \mathsf{Pizza} \\
i_M \in & \mathsf{Mozzarella} \\
i_T \in & \mathsf{Tomato} \\
\langle i_P, i_M \rangle \in & \mathsf{hasTopping} \\
\langle i_P, i_T \rangle \in & \mathsf{hasTopping} \\
\mathcal{O} \not\models & i_P \in \mathsf{Margherita}
\end{aligned}
$$

Figure 7.1: OWL axioms showing the need for a covering axiom.

provide the implied price for a defined pizza. In general, if there is an OWL class $A \sqsubseteq D$: "1", and an OWL individual $a \in A$, the OWL reasoners conforming the OWL API return an empty set to a query for the values of D for the individual a. Hence, the value of the property has to be gathered manually by analysing the ontology. This is a perfect example for a custom mapping. We developed a mapping which analyses the explicit structure of all types of an OWL individual. Thus, it adds the value "1" for the property D to the inferred properties of an OwlFrame assigned to a. Both use cases demonstrate the power and elegance of the separated mapping in the Mooop approach: It offers a full access to the OWL model but hides the complex integration from most Java developers who are only working with hybrid classes.

The the price of a custom pizza is: (1) the explicit price of the pizza if the custom pizza matches a predefined one, or (2) the sum of the explicit prices of the toppings. This could be implemented by checking whether the current custom pizza has an explicit price in the ontology or not. However, the more elegant way is to define an OWL class for all pizzas with a price and make use of the dynamic method dispatch facility of Mooop. The result can be seen in Listing 7.4: The default price calculation method, calculatePrice(), sums up the prices of the toppings, but if the pizza is a PricedPizza, that is a pizza with a defined price, then the protected method getPrice() is called, which, itself, is bound to the OWL property hasPrice.

Another requirement which has to be implemented is that is has to be ensured that a pizza is topped with at most one topping of each kind. Hence, a pizza with two ham toppings should be illegal and impossible to create. This restriction could be expressed in OWL, however, it is very complex because this has to be defined for each and every topping: this could look like $\mathsf{HamPizza} \sqsubseteq \exists \mathsf{hasTopping}.\mathsf{Ham} \sqcap \leqslant 1\mathsf{hasTopping}$. However, in Java such an invariant is easy to check by some behaviour. In this case, the developer decided that the invariant check is part of the specific business logic encapsulated in the Pizza class. Therefore, the check has been implemented in the method addTopping(...) as shown in Listing 7.5: It checks for each existing topping of the pizza whether the asserted types are equal to the asserted types of the new topping,

```
@HybridClass("Pizza")
public class Pizza {
  @OwlDispatch({ @OwlDispatchEntry(owlType = "PricedPizza", methodName =
      "getPrice") })
  public Integer calculatePrice() {
    int price = 0;
    // sum up prices of the toppings
    for (Topping topping : getToppings()) {
      price += topping.getPrice();
    }
    return price;
  }
  @OwlProperty(value = "hasPrice", valueType = Integer.class)
  protected Integer getPrice() {
    return null; // will not be executed
  }
  ...
}
```

Listing 7.4: Example for the usage of the OWL model-based method dispatch for calculating the price of a pizza.

and adds the new topping if there is non equal topping already existent. Notice that the method addTopping_internal(...) actually performs the modification of the assigned OwlFrame.

```
@HybridClass("Pizza")
public class Pizza {
  public void addTopping(final Topping topping) {
    boolean isAlreadyAdded = false;
    for (Topping ownTopping : getToppings()) {
      // check whether the toppings have the same type, i.e., are equal
      if (ownTopping.getAssertedTypes().equals(topping.getAssertedTypes
          ())) {
        isAlreadyAdded = true;
        break;
      }
    }
    if (!isAlreadyAdded) {
      addTopping_internal(topping);
    }
  }
  @OwlProperty("hasTopping")
  protected void addTopping_internal(final Topping topping) {
    // will not be executed
  }
  ...
}
```

Listing 7.5: The business logic ensuring that at most one topping of each kind is added to a pizza.

At this point, the domain model is ready to be used by the application. However, there are some further interesting information in the OWL model which can be useful, e.g., a pizza can have a spiciness defined by a property hasSpiciness, and a pizza can originate from a special country defined by the property hasCountryOfOrigin. However, not all pizzas have these properties. Hence, the access to this information can be implemented as indirect properties. Listing 7.6 shows the method for accessing the properties of the pizza indirectly. Thereby, the result is immutable, i.e., the set and its entries can not be changed. Since the pizza does not offer any methods for changing properties other than

hasTopping, there is not way to manipulate the information.

```
@HybridClass("Pizza")
public class Pizza {
  @OwlProperty()
  public Map<String, Set<Object>> getProperties() {
    return null; // will not be executed
  }
  ...
}
```

Listing 7.6: The method for accessing the properties of a pizza indirectly.

The name of the predefined pizzas is defined as their class name. Therefore, in order to tell the user which pizza he just configured, its necessary to get the type of a pizza which is a subclass of NamedPizza, the superclass of all predefined pizzas. Since the specific type of a pizza is never set, this type has to be an inferred type. Unfortunately, an OwlFrame with the asserted type Pizza is also always inferred to be an instance of several types, e.g., Thing and Food. Therefore, we need a special binding which returns only the types of an OwlFrame which are a subtype of NamedPizza. Unfortunately, there is no predefined binding for this, but Mooop allows the easy definition of a custom binding for this. The Java annotation can be seen in Listing 7.7. Additionally, the listing shows the method handler which is executed for a method annotated with @PizzaType. This demonstrates how simple it is to adapt the Mooop framework to the needs of a project.

Finally, the pizza configurator is finished and has all its functionality. Listing 7.8 shows a workflow which demonstrates some of the capabilities.

7.1.2 Medical Patient Model

The medical patient model has already been introduced in Chapter 4.2.2: it is based on a case study for the Core Model-Builder hybrid integration framework and requires a sophisticated hybrid integration approach. Therefore, it is well qualified to exemplify important design features of the Mooop concept. In the following we outline a prototypical Mooop-based implementation of the medical patient model which allows to perform the workflow shown in Chapter 4.2.2.

The main concept of the medical patient model is the ClinicalProblemGlimpse Java class shown in Listing 7.9. It represents the results of a single medical examination of a patient: it has a type which represents the general kind of problem, e.g., Cancer, a location which states where the problem is, e.g., the breast, and descriptors which are indirect accessible properties describing the problem in more detail, e.g., a stage of the medical condition. There are further hybrid classes like Locus or GenericMedicalEntity which have been implemented for the case study. However, they are much simpler and, therefore, not presented here. Please refer to the source code and Appendix D for more details on them.

The medical patient model application uses a custom binding for accessing the properties: @MedicalProperty. The corresponding method handler, MedicalPropertyHandler, extends the Mooop OwlPropertyHandler with two features which are also provided by the original Patient Chronicle Model: first, the pos-

```
@MooopMethod(PizzaTypeHandler.class)
public @interface PizzaType {
}

public class PizzaTypeHandler implements MethodHandler {
  /** The base class the pizza type is a subclass of. */
  private static final String PIZZA_BASE_CLASS = "NamedPizza";
  ...
  @Override
  public Object call(final Object hybridObject, final Object[] arguments
      , final MethodInvocator originalMethod) {
    // go through all types and look for a subclass of the base class.
    OWLClassExpression result = null;
    // first try asserted types for performance reasons
    for (OWLClassExpression type : owlFrame.getAssertedTypes()) {
      if (reasoningMapping.isSubType(type, namedPizzaType) && !type.
          equals(namedPizzaType)) {
        result = type;
        break;
      }
    }
    if (result == null) {
      // second try inferred types
      ...
    }
    return result != null ? frameDomainConverter.
        convertOwlClassExpression(result, returnType) : null;
  }
  @Override
  public void initialize(final Class<?> clazz, final Method method,
      final OwlFrame owlFrame, final MooopConfiguration
      mooopConfiguration) throws MooopConfigurationException {
    ...
    // get OWL class of pizza base class
    namedPizzaType = mooopConfiguration.getDomainFrameConverter().
        convertOwlClassExpression(PIZZA_BASE_CLASS);
    ...
  }
}
```

Listing 7.7: Annotation and Method handler for the @PizzaType annotation.

```
@Test
public void PizzaWorkflow2() throws MooopConfigurationException,
    MooopParserException, OWLOntologyCreationException {
  ...
  // create and add mozzarella topping
  Topping mozzarella = mooopManager.create(Topping.class);
  mozzarella.addAssertedType("Mozzarella");
  pizza.addTopping(mozzarella);
  // classify
  pizza.classify();
  // check
  assertNull(pizza.getType());
  assertEquals(Integer.valueOf(2), pizza.calculatePrice());
  ...
  // load and add tomato topping
  Topping tomato = mooopManager.load("tomato", Topping.class);
  assertArrayEqualsIgnoringOrder(new String[] { "Tomato", "PizzaTopping"
      }, tomato.getAssertedTypes());
  pizza.addTopping(tomato);
  // classify
  pizza.classify();
  // check
  assertEquals("Margherita", pizza.getType());
  assertEquals(Integer.valueOf(5), pizza.calculatePrice());
  assertArrayEqualsIgnoringOrder(new Thing[] { mooopManager.load("mild",
      Thing.class) }, pizza.getProperties().get("hasSpiciness"));
}
```

Listing 7.8: Final workflow in the pizza configurator case study.

```
@HybridClass("ClinicalProblem")
public class ClinicalProblemGlimpse {
  @MedicalProperty()
  public Map<String, GenericMedicalEntity> getDescriptors() {
    return null; // will not be executed
  }
  @MedicalProperty(value = "has_locus", valueType = Locus.class)
  public Locus getLocation() {
    return null; // will not be executed
  }
  @OwlType(mostSpecific = true)
  public String getType() {
    return null; // will not be executed
  }
  @MedicalProperty(methodType = MethodType.ADD)
  public void setDescriptor(final String string, final Object value) {
    // will not be executed
  }
  @MedicalProperty(value = "has_locus", methodType = MethodType.ADD)
  public void setLocation(final Locus locus) {
    // will not be executed
  }
  @OwlType(methodType = MethodType.ADD)
  public void setType(final String string) {
    // will not be executed
  }
}
```

Listing 7.9: The hybrid class ClinicalProblemGlimpse.

sibility to hide designated properties, i.e., they are not accessible at all, and to omit the directly accessible properties of a hybrid class in the map of indirect properties, e.g., the property has_locus is not in the map returned by getDescriptors(). Especially the latter feature seems to be very useful and reasonable but has not been considered while implementing the Mooop framework. However, the flexible and customizable Mooop approach allows for an easy extension of the binding and, hence, can mimic the behaviour of the original Core Model-Builder. This exemplifies the generality of the Mooop concept.

However, there are more complex requirements on the mapping for the medical patient model: In contrast to the OWL individual-based pizza configurator, the original Patient Chronicle Model is based on OWL classes. Hence, the user does not create and classify an OWL individual while working with the application, but instead creates and classifies a complex OWL class. Besides the pros and cons of this approach, it is important to notice that this feature is used by other applications, especially ontology editors (see Chapter 4.1), as well.

Although the OwlFrame is tightly connected to an OWL individual, the MedicalMapping implements an OWL class-based classification. This is possible, since the mapping can create an OWL individual which is never added to the ontology. Therefore, it can be seen as faked. The same is done for the asserted types and properties: they are saved in the OwlFrame but never written to the ontology. If the OwlFrame queries the mapping for its inferred types or properties, they are computed using a complex process: the asserted types and properties of the OwlFrame are transformed into a complex OWL class which is, subsequently, classified. The inferred properties are gathered from the class definitions of the inferred types. Using this complex mapping and binding, we have implemented the workflow from Chapter 4.2.2 as depicted in Listing 7.10.

The sanctioning employed by the Patient Chronicle Model is far more complex than the description above[1]: for instance, there is a special rule identifying concepts as the units of numbers. Although the medical patient model does not implement this elaborated sanctioning of the Patient Chronicle Model, we are, however, confident this is achievable by extending the current MedicalMapping. This exemplifies that the Mooop concept is suited for complex applications like the Patient Chronicle Model.

The medical patient model and the Patient Chronicle Model allows for a comparison of the two hybrid integration approaches which have been employed: Core Model-Builder and Mooop. However, this comparison is not easy since the Core Model-Builder allows to integrate arbitrary models into Java but Mooop is specialised on OWL models. Hence, a comparison of the OWL plug-in (which contains the Patient Chronicle Model sanctioning) of the Core Model-Builder and the MedicalMapping is not sensible. Taking the results of the medical patient model into account, we think, however, that the expressible integration semantics is equally great and both approaches allow the development of complex ontology-based applications. A further comparison of both approaches concerning the integration in the Java model reveals advantages and shortcomings of the current Mooop prototype. On the one hand, Mooop is missing useful query functions. The Core Model-Builder offers sophisticated functions for querying

[1] However, to the best of our knowledge, there exists no comprehensible documentation about the sanctioning.

```
@Test
public void doWorkflow() throws MooopConfigurationException {
  ClinicalProblemGlimpse clinicalProblemGlimpse = mooopManager.create(
      ClinicalProblemGlimpse.class);
  assertEquals("ClinicalProblem", clinicalProblemGlimpse.getType());
  // set type to cancer
  clinicalProblemGlimpse.setType("Cancer");
  assertEquals("Cancer", clinicalProblemGlimpse.getType());
  assertEquals("Stage_status", clinicalProblemGlimpse.getDescriptors().
      get("has_stage_status").getType());
  // set location Breast
  Locus locus = mooopManager.create(Locus.class);
  locus.setType("Breast");
  clinicalProblemGlimpse.setLocation(locus);
  assertEquals("Breast", clinicalProblemGlimpse.getLocation().getType())
      ;
  assertEquals("Breast_Cancer", clinicalProblemGlimpse.getType());
  // set stage III
  GenericMedicalEntity stageIII = mooopManager.create(
      GenericMedicalEntity.class);
  stageIII.addAssertedType("StageIII");
  clinicalProblemGlimpse.setDescriptor("has_stage_status", stageIII);
  assertEquals("Breast", clinicalProblemGlimpse.getLocation().getType())
      ;
  assertEquals("StageIII", clinicalProblemGlimpse.getDescriptors().get("
      has_stage_status").getType());
  assertEquals("Breast_cancer_stage_III", clinicalProblemGlimpse.getType
      ());
  assertEquals("Sub_stageA\n or Sub_stageB\n or Sub_stageC",
      clinicalProblemGlimpse.getDescriptors().get("has_sub_stage_status
      ").getType());
  // set type to leukemia
  clinicalProblemGlimpse.setType("Leukemia");
  assertEquals("Leukemia", clinicalProblemGlimpse.getType());
  assertEquals("Hematopoietic_System", clinicalProblemGlimpse.
      getLocation().getType());
  assertFalse(clinicalProblemGlimpse.getDescriptors().containsKey("
      has_sub_stage_status"));
}
```

Listing 7.10: The implementation of the workflow from Chapter 4.2.2.

possible values for a property. This is a useful feature which can prevent errors since the values of properties do not have to be set blind. Therefore, this is a reasonable extension of Mooop. On the other hand, we think that the declarative definition of the hybrid classes in Mooop is a great advantage. The definition of hybrid classes in the Core Model-Builder has to be performed by extending classes and implementing abstract methods. This mixes the code for the integration with code for the functionality of the model and makes the code harder to understand. The Mooop binding based on Java annotations allows for a clear separation of both concerns and, thus, does not obscure the structure of the model entities. Furthermore, since the bindings in Mooop can be flexibly customised, one can easily implement a new integration functionality.

7.2 Validation of Requirements

In this chapter we show that the Mooop concept fulfils the requirements on a generic integration approach as introduced in Chapter 4.3. These are: (1) offering a hybrid integration approach, (2) allow for the customisation of the integration, and (3) provide means for a declarative definition of hybrid classes.

This validation is mainly based on the concept implemented by the prototype and the case studies exploiting the features of Mooop.

Mooop was developed with the aim to create a customisable, flexible and declarative hybrid integration approach. Therefore, it fulfils the requirement for a hybrid integration by design. Mooop offers the possibility to define directly and indirectly represented types and properties of hybrid classes. Thus, it is possible to preserve key features of Java: directly represented types and properties preserve the type safety, the hybrid classes allow the definition of a domain-specific API for the OWL model, and the behaviour can be encapsulated within the hybrid classes. On the other hand, the hybrid approach can also preserve key OWL features. Indirectly represented types allow a dynamic and multiple typing of the OWL entities. Furthermore, indirect properties enable a lax object conformance and their runtime dynamics allow the reasoning to change the structure of the OWL entity. Notice that the degree of the OO-likeness or OWL-likeness of the integrated model can be adjusted whereby more OO-likeness means less OWL-likeness and vice versa. It can be even adjusted to the extremes: complete direct or indirect integration. This leads to the insight that the hybrid modelling approach of Mooop subsumes direct and indirect modelling to some degree: If one uses solely hybrid classes with a direct bound type and direct properties for modelling the result is a Java model which directly integrates an OWL model. On the contrary, if one defines a model solely based on indirect types and properties, the result will probably be a model with only one hybrid class which can be seen as a copy of the OwlFrame. However, this is actually a Java model which indirectly integrates an OWL model.

Mooop provides the users with a two means to customize the integration performed by the framework: the mapping and the binding. The mapping implements the custom link between the OWL model and the OwlFrame. Therefore, it is the component which should implement the sanctioning and general manipulation mechanism for the OWL model. The binding facility of Mooop, on the other hand, allows to define new bindings which access the information of an OwlFrame in a specific way. Both provide a high degree of flexibility which is mainly limited by the structure of the OwlFrame. However, we think that the presented structure of the OwlFrame is very generic and allows for a wide range of applications.

For a Java developer using the Mooop framework, the most remarkable thing are the binding annotations. The framework seems to be completely declarative to the developer who does not extend Mooop. This has the immense advantage that the developer can easily distinguish the binding code and the functional code of a hybrid class. This eases his work and makes it less error-prone. However, Mooop can not be solely used in a declarative manner. If the predefined mappings and bindings do not suit the needs of a project, a specialised developer has to write custom ones. This task is currently not declarative. We think that it is actually hard to accomplish this goal because the definition still has to be very expressive and flexible. However, this problem is neglectable since the development effort of the mapping and binding should be a fraction of the complete project effort.

We showed that the concept of Mooop in general and the prototype in special are fulfilling the requirement for a generic integration of OWL into Java.

Performance has not been an issue, so far. However, in the following we want to present some results of an ad hoc analysis of the workflows for the case studies. The main insight is that performance is a main problem for ontology-based applications. Even the simple workflow from Listing 7.8 takes around 3.9 seconds on a our test system, an Intel T1300 notebook with a clock speed of 1.66 GHz and 3 GB main memory. Notice that the OWL ontology and the Java model are small compared to real applications. A further analysis with a profiler reveals that more than 91 per cent of the time is consumed by the OWL API and the OWL reasoner (see Appendix D). There are two implications from this: first, the research on improving OWL reasoners is of immense importance. One promising approach is incremental reasoning, i.e., not the complete ontology is reclassified after a modification of the ontology but only the fraction that is affected by the change. Second, the mapping should use a reasoner as seldom as possible. Hence, the developer of the mapping and binding should be experts in both OWL and Java, and should be able to analyse the performance implications of code.

7.3 Methodological Considerations

The Mooop concept allows the definition of hybrid models integrating OWL and Java. This is a complex task because the developer of the models has to fell a lot of hard decisions: which information should be put in the OWL model and which in the Java model? Which top level concepts should be directly represented? What direct properties does a hybrid class have? This is just a selection but it make obvious that some kind of methodology is needed which provides a structured decision process for these issues. During the analysis of different modelling approaches, Puleston et al. derived some methodological considerations concerning the development of direct, indirect, and hybrid models [59]. Their ideas apply to the modelling with Mooop as well. However, our case studies reveal that the Mooop framework demands further guidelines especially concerning the development of the mapping and binding. In the following, we analyse several Mooop specific integration issues which demand a sophisticated decision process. We have not developed a methodology but we present some ideas from the case studies.

The Mooop mapping and the modelling guideline for the ontology are very interrelated: they depend on and define each other. The developer has to decide for a modelling guideline and, thereby, influences the mapping. Two scenarios can be distinguished. First, the OWL model is given because the application is based on some shared ontology like GALEN. Hence, the modelling guideline is already defined either explicit or implicit. In this case, the developer has to develop a mapping which translates the knowledge from the OWL model into an OwlFrame based on the given guideline. If there are standard modelling approaches, then it is possible that a predefined mapping can be reused and, as the case may be, refined. Otherwise, it is advantageous if the modelling guideline is explicitly defined because then the mapping can be easily encoded. On the contrary, if the guideline is only implicitly encode in the ontology itself then an iterative approach may suit: the developer refines an initially simple mapping stepwise if new mapping requirements come up. In the second sce-

nario, the ontology is developed together with the application. In this case the developer has to decide whether to use a common modelling guideline for which a predefined mapping exists, or use a custom modelling guideline and write a custom mapping. This is a complex issue and corresponding decision guidances still have to be researched.

The extraction of the integration logic from the whole application logic allows a clean separation of concerns in the development of ontology-based applications. Even more, it allows for a separation of specialised developer roles in the project. Application developers are Java experts designing the Java model and using the binding to define integration points for the OWL model. On the other hand, the specialised mapping developer is an expert in Java, OWL, Mooop, and the OWL API. He develops the complex mapping and custom bindings which are used by the application developers. Therefore, the complexity of the integration and OWL is hidden from the application developers and only visible to a few mapping developers.

The distinction of integration and application logic, on the other hand, imposes the duty to split it up. The issue is to decide which part of the behaviour is integration logic, i.e., part of the mapping, and which one is application logic, i.e., part of the binding or hybrid classes. As a rule of thumb, every functionality which needs a direct access to the ontology and cross-section functionality which is necessary for the whole integration should be implemented in the mapping. An example for this is a sanctioning mechanism which derives a static structure for an OWL individual from its type and property assertions. On the other hand, functionality which is very specific for a certain concept should be put into a custom binding or implemented in the methods of a hybrid class. For instance, the requirement for an automatic object classification for objects of a specific hybrid class should be implemented by using a specialised binding. However, this guidance is just a starting point for the development of a complete methodology. This is open to future research.

Chapter 8

Conclusion

This work presented Mooop: a generic approach to integrate ontological (OWL) into object-oriented (Java) models. In the following, we summarise the concept of Mooop and outline its advantages, and point out directions for future work.

8.1 Summary

Exploiting ontologies in object-oriented applications is a difficult task because both modelling approaches are very different and are separated by an impedance mismatch which is hard to overcome. Common direct or indirect integration approaches have severe shortcomings: either the OWL features which can be used are limited or the resulting combined model is hard to use. As a hybrid integration approach, Mooop exploits the strength of direct and indirect integration and, thus, is both powerful and easy to use.

In contrast to other hybrid integration approaches, Mooop explicitly divides the integration into three layers. First, the OwlFrame defines a generic indirect Java model for ontological knowledge. Therefore, it can be flexibly used in a variety of contexts. Furthermore, it has been designed with the goal to be dynamic and, therefore, it enables a powerful reasoning. Second, the mapping defines a custom and project specific translation between the OWL model and the OwlFrame. It defines how knowledge from the ontology is integrated into the object-oriented application. Third, the binding allows the definition of hybrid classes, i.e., Java model classes representing OWL concepts directly and indirectly. The binding is, therefore, used to define a domain specific API for accessing the information from the OwlFrame. The definition of the hybrid classes is performed in a declarative manner through Java annotations. The Mooop concept provides a clear separation of concerns and a great flexibility since both the mapping and binding can be customized to suit the needs of a specific project. We have showed this on two case studies and, additionally, outlined some methodological considerations for developing applications with Mooop.

The basic idea of Mooop, i.e., a hybrid integration of OWL into Java and a customizable mapping and binding facility, is a promising approach for a wide

range of applications: on the one hand, the hybrid integration enables a powerful and yet domain specific access to the ontological information. On the other hand, the customizable mapping and binding enable the easy adjustment of the integration to the requirements and modelling guideline of a specific project. This includes the possibility to adjust the degree of indirectness or directness of the integration. Furthermore, the design of the mapping and binding facility allows to combine several mappings or bindings to create new ones. Additionally, the binding is defined through declarative Java annotations which fosters the separation of infrastructural code, i.e., code defining the integration, and functional code, i.e., code that expresses specific application features. Both facts facilitate reuse and, hence, enables shorter development cycles and less code which leads to fever bugs and an improved maintainability of the application. We can even imagine repositories for sharing mappings and bindings which are aligned with a specific OWL modelling approach.

8.2 Outlook

Mooop is an interesting concept for the integration of OWL ontologies into Java models. Although the prototype is a working implementation of the approach, it is not supposed to be used in real projects yet. There is still research to be done concerning two aspects of Mooop: the functionality and the methodology.

Throughout this work we already pointed to interesting extensions of the current concept. Thereby, the development of more sophisticated mappings and bindings is probably the most pressing issue. For instance, mappings for standard OWL modelling guidelines could extend the reuse, structured mappings could ease the integration of external sanctioning facilities like the *reasonableness layer* [9], and a generic SPARQL query facility like the one in agogo (see Chapter 3.1) could extend Mooop to handle ontology design patterns. Furthermore, the case studies revealed the need for a more elaborated query facility that allows to query the OWL model for possible values (OwlFrames) or value types (OWL classes) for a property. Another important subject which has not been researched yet is how the runtime model of a Mooop system can be persisted. It seems obvious that a special concept is necessary since hybrid objects contain both a Java state and a OWL state. We think that it is most promising to separate the issues of persisting the runtime OWL model, i.e., the OWL individuals created throughout the execution of the application, and persisting the Java state of hybrid objects. However, when the model is loaded is has to be ensured that both parts are correctly merged again. This way, the user can choose the best persistence solution for both models, e.g., a RDF triple store for the OWL model and a relational database for the Java model.

We have outlined some considerations concerning a methodology for developing applications with Mooop. However, these are only a starting point. To the best of our knowledge, there is no elaborated modelling methodology for hybrid models which provides guidance for choosing the directly integrated top-level concepts and proposes a division between the two integrated models. Such a methodology would be even more valuable if it is not specific for Mooop but applies to all hybrid integration approaches. However, Mooop also lacks a specific methodology for guiding the developer how to choose a modelling guideline

and develop a matching mapping for it, and how to separate the application logic from the integration logic.

All former issues are related to the concept of Mooop presented in this work. However, we can imagine to leverage further advantages of the approach if it is integrated into other methodologies. Model Driven Architecture (MDA) [43] is a software development approach which puts application models into focus. It proposes a workflow where the software development starts with an abstract, high level model of the application and refines it in subsequent steps. TwoUse (see Chapter 3.3) is an MDA approach for the joint modelling of an ontology and an object-oriented application. It is a powerful and holistic development method, but lacks several features concerning the OWL-OO integration. We think that a combination of TwoUse and Mooop could lead to a powerful and fully-fledged development framework for ontology-based applications and, therefore, is an intriguing future research topic.

Finally, we have outlined that the basic idea of Mooop, i.e., the division of the OWL-OO integration into a mapping and binding, can be generalised. This general approach for integrating models of different expressiveness is promising. However, it has not proved its feasibility and advantages yet. Therefore, it could be an interesting future research subject to design, implement, and use such a general concept for, e.g., integrating XML schema into Java.

Bibliography

[1] Pellet Features. accessed 22 Sep 2010. URL `http://clarkparsia.com/pellet/features`.

[2] A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications (Technical Supplement). accessed 24 Oct 2010, 2008. URL `http://intranet.cs.man.ac.uk/bhig/clef_misc/Ontology-Backed-Model-Builder-Supplement.pdf`.

[3] The Protégé Ontology Editor and Knowledge Acquisition System. accessed 23 Aug 2010, 2009. URL `http://protege.stanford.edu`.

[4] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. Ontologies, Metamodels, and the Model-Driven Paradigm. *Ontologies for Software Engineering and Software Technology*, pages 249–273, 2006.

[5] Franz Baader and Werner Nutt. Basic Description Logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 43–95. Cambridge University Press, New York, NY, USA, 2003.

[6] Patricia G. Baker, Andy Brass, Sean Bechhofer, Carole A. Goble, Norman W. Paton, and Robert Stevens. TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources. In *Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology (ISMB'98)*, pages 25–34. AAAI Press, July 1998.

[7] Juozas Baliuka, Chris Nokleberg, Matas Ramoska, and Wes Biggs. Code Generation Library. accessed 24 Sep 2010. URL `http://cglib.sourceforge.net/`.

[8] Sean Bechhofer and Carole A. Goble. Using a Description Logic to Drive Query Interfaces. In Ronald J. Brachman, Francesco M. Donini, Enrico Franconi, Ian Horrocks, Alon Y. Levy, and Marie-Christine Rousset, editors, *Description Logics*, volume 410 of *URA-CNRS*, 1997.

[9] Sean Bechhofer and Ian Horrocks. Driving User Interfaces from FaCT. In Franz Baader and Ulrike Sattler, editors, *Description Logics*, volume 33 of *CEUR Workshop Proceedings*, pages 45–54. CEUR-WS.org, 2000.

[10] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[11] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.98. Technical Report Marco Polo Edition, The Friend of a Friend (FOAF) project, August 2009.

[12] Keith L. Clark and Frank G. McCabe. Ontology oriented programming in Go! *Appl. Intell.*, 24(3):189–204, 2006.

[13] Taylor Cowan. Jenabean: Easily bind JavaBeans to RDF. accessed 24 Oct 2010. URL `http://www.ibm.com/developerworks/java/library/j-jenabean.html`.

[14] Ian Dickinson. Jena Ontology API. accessed 19 Sep 2010, February 2009. URL `http://jena.sourceforge.net/ontology/index.html`.

[15] Andreas Eberhart. Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2002.

[16] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[17] Martin Fowler. Analysis Patterns 2. accessed 23 Aug 2010, Jun 2000. URL `http://martinfowler.com/apsupp/accby.pdf`.

[18] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 edition, November 1994.

[19] Neil M. Goldman. Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 850–865. Springer, 2003.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. The Java Series. Addison-Wesley, third edition, May 2005.

[21] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, and Ulrike Sattler. OWL 2: The Next Step for OWL. *J. Web Sem.*, 6(4):309–322, 2008.

[22] Hans-Jörg Happel and Stefan Seedorf. Applications of Ontologies in Software Engineering. In *International Workshop on Semantic Web Enabled Software Engineering (SWESE'06)*, Athens, USA, November 2006.

[23] William H. Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA*, pages 411–428, 1993.

[24] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3c recommendation, W3C, October 2009.

[25] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[26] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax. W3C note, W3C, October 2009.

[27] Ian Horrocks and Peter F. Patel-Schneider. Three Theses of Representation in the Semantic Web. In *WWW*, pages 39–47, 2003.

[28] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *J. of Web Semantics*, 1(1):7–26, 2003.

[29] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin N. Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3c member submission, W3C, May 2004.

[30] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible $\mathcal{SROIQ}$. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 57–67. AAAI Press, 2006.

[31] ISO and IEC. ISO/IEC 13250 Topic Maps. Technical Report 13250, International Organization for Standardization / International Electrotechnical Commission, May 2002.

[32] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. Automatic Mapping of OWL Ontologies into Java. In Frank Maurer and Günther Ruhe, editors, *SEKE*, pages 98–103, 2004.

[33] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James A. Hendler. Swoop: A Web Ontology Editing Browser. *J. Web Sem.*, 4(2):144–153, 2006.

[34] Günter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In Rachid Guerraoui, editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 1999.

[35] Holger Knublauch. Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL. In David S. Frankel, Elisa F. Kendall, and Deborah L. McGuinness, editors, *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[36] Holger Knublauch, Daniel Oberle, Phil Tetlow, and Evan Wallace. A Semantic Web Primer for Object-Oriented Software Developers. W3C Working Group Note 9 March 2006, W3C, 03 2006.

[37] Seiji Koide and Hideaki Takeda. OWL-Full Reasoning from an Object Oriented Perspective. In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *ASWC*, volume 4185 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[38] Seiji Koide, Jans Aasman, and Steve Haflich. OWL vs. Object Oriented Programming. In *Workshop on Semantic Web Enabled Software Engineering (SWESE)*, November 2005. Galway, Ireland.

[39] James Leigh. Elmo User Guide. accessed 24 Aug 2010, 2010. URL `http://www.openrdf.org/doc/elmo/1.5/user-guide/index.html`.

[40] Barbara Liskov. Data Abstraction and Hierarchy. *OOPSLA'87 Proceedings, Addendum to the ...*, 23(5):17–34, May 1988.

[41] Frank Manola and Eric Miller. RDF Primer. W3c recommendation, W3C, February 2004.

[42] Marvin Minsky. A Framework for Representing Knowledge. (MIT) Artificial Intelligence Memo 306, Department of Computer Science, Massachusettes Institute of Technology, June 1974.

[43] OMG. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, Object Management Group, June 2003.

[44] OMG. Ontology Definition Metamodel, Version 1.0. Technical Report formal/2009-05-01, Object Management Group, May 2009.

[45] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.3, without change bars. Technical Report formal/2010-05-03, Object Management Group, May 2010.

[46] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3, without change bars. Technical Report formal/2010-05-05, Object Management Group, May 2010.

[47] Tope Omitola, Christos L. Koumenides, Igor O. Popov, Yang Yang, Manuel Salvadores, Martin Szomszor, Tim Berners-Lee, Nicholas Gibbins, Wendy Hall, m. c. schraefel m. c. schraefel, and Nigel Shadbolt. Put in Your Postcode, Out Comes the Data: A Case Study. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *ESWC (1)*, volume 6088 of *Lecture Notes in Computer Science*, pages 318–332. Springer, 2010.

[48] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. ActiveRDF: Object-Oriented Semantic Web Programming. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 817–824. ACM, 2007.

[49] Eyal Oren, Benjamin Heitmann, and Stefan Decker. ActiveRDF: Embedding Semantic Web data into object-oriented languages. *J. Web Sem.*, 6 (3):191–202, 2008.

[50] International Health Terminology Standards Development Organisation. SNOMED CT. accessed 31 Aug 2010, 2010. URL `http://www.ihtsdo.org/snomed-ct/`.

[51] International Health Terminology Standards Development Organisation. SNOMED Clinical Terms User Guide. Technical Report CVR #: 30363434, International Health Terminology Standards Development Organisation, January 2010.

[52] Alexander Paar. Zhi# - Programming Language Inherent Support for Ontologies. In Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter, editors, *ateM '07: Proceedings of the 4th International Workshop on Software Language Engineering*, pages 165–181, Mainz, October 2007. Mainzer Informatik-Berichte.

[53] Fernando Silva Parreiras and Steffen Staab. Using Ontologies with UML Class-based Modeling: The TwoUse Approach. *Data and Knowledge Engineering*, 2010.

[54] Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. APIs à gogo: Automatic Generation of Ontology APIs. In *ICSC*, pages 342–348. IEEE Computer Society, 2009.

[55] Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging OWL Ontologies. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 633–640. ACM, 2005.

[56] A. Passant. FOAFMap: Web 2.0 meets the Semantic Web. In *Proc. Scripting Challenge, ESWC2006 Workshop on Scripting for the Semantic Web*. Citeseer, 2006.

[57] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C recommendation, W3C, January 2008.

[58] Colin Puleston, James Cunningham, and Alan L. Rector. A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications. In *OWL: Experiences and Directions (OWLED)*, page online, 2008.

[59] Colin Puleston, Bijan Parsia, James Cunningham, and Alan L. Rector. Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[60] Franz Puntigam. Skriptum zu Objektorientierte Programmierung Wintersemester 2010/2011. accessed 11 Oct 2010, 2010. URL `http://www.complang.tuwien.ac.at/franz/objektorientiert/skript10-1seitig.pdf`.

[61] Alan L. Rector, Jeremy Rogers, and P. Pole. The GALEN High Level Ontology. In *MIE*, volume 96, pages 174–178, 1996.

[62] Jeff Rothenberg. The Nature of Modeling. *AI, Simulation & Modeling*, pages 75–92, August 1989.

[63] Ulrike Sattler, Diego Calvanese, and Ralf Molitor. Relationships with other Formalisms. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 137–177. Cambridge University Press, New York, NY, USA, 2003.

[64] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[65] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *OWLED*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[66] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2): 51–53, 2007.

[67] Thomas Springer and Anni-Yasmin Turhan. Employing Description Logics in Ambient Intelligence for Modeling and Reasoning about Complex Situations. *Journal of Ambient Intelligence and Smart Environments*, 1(3): 235–259, 2009.

[68] Henry Story. So(m)mer: Semantic Object (Metadata) MappeR. accessed 25 Aug 2010, 2010. URL `https://sommer.dev.java.net`.

[69] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge Engineering: Principles and Methods. *Data Knowl. Eng.*, 25(1-2):161–197, 1998.

[70] Ben Szekely and Joe Betz. Jastor – Typesafe, Ontology Driven RDF Access from Java. accessed 24 Aug 2010, 2010. URL `http://jastor.sourceforge.net/`.

[71] Thanh Tran, Holger Lewen, and Peter Haase. On the Role and Application of Ontologies in Information Systems. In *RIVF*, pages 14–21. IEEE, 2007.

[72] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 292–297. Springer, 2006.

[73] Max Völkel and York Sure. RDFReactor – From Ontologies to Programmatic Data Access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, 2005.

[74] C. Wroe. Is Semantic Web technology ready for Healthcare? In *3rd European Semantic Web Conference (ESWC 06), Budva, Montenegro*, 2006.

[75] Joseph W. Yoder, Federico Balaguer, and Ralph E. Johnson. Architecture and Design of Adaptive Object Models. *SIGPLAN Notices*, 36(12):50–60, 2001.

Appendix A

Description Logics Syntax and Semantics

The following tables are describing the syntax and semantics of the Description Logic used throughout this work. They are taken from [28].

Abstract Syntax	DL Syntax	Semantics
Class(A partial C_1 ...C_n)	$A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
Class(A complete C_1 ...C_n)	$A = C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
EnumeratedClass(A o_1 ...o_n)	$A = \{o_1, \ldots, o_n\}$	$A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
SubClassOf(C_1 C_2)	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
EquivalentClasses(C_1 ...C_n)	$C_1 = \ldots = C_n$	$C_1^{\mathcal{I}} = \ldots = C_n^{\mathcal{I}}$
DisjointClasses(C_1 ...C_n)	$C_i \sqcap C_j = \bot, i \neq j$	$C_i^{\mathcal{I}} \cap C_j^{\mathcal{I}}\{\}, i \neq j$
Datatype(D)		$D^{\mathcal{I}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
DatatypeProperty(U super(U_1)...super(U_n)	$U \sqsubseteq U_i$	$U^{\mathcal{I}} \subseteq U_i^{\mathcal{I}}$
domain(C_1) ...domain(C_m)	$\geqslant 1\, U \sqsubseteq C_i$	$U^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
range(D_1) ...range(D_l)	$\top \sqsubseteq \forall U.D_i$	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times D_i^{\mathcal{I}}$
[Functional])	$\top \sqsubseteq \leqslant 1\, U$	$U^{\mathcal{I}}$ is functional
SubPropertyOf(U_1 U_2)	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
EquivalentProperties(U_1 ...U_n)	$U_1 = \ldots = U_n$	$U_1^{\mathcal{I}} = \ldots = U_n^{\mathcal{I}}$
ObjectProperty(R super(R_1)...super(R_n)	$R \sqsubseteq R_i$	$R^{\mathcal{I}} \subseteq R_i^{\mathcal{I}}$
domain(C_1) ...domain(C_m)	$\geqslant 1\, R \sqsubseteq C_i$	$R^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
range(C_1) ...range(C_l)	$\top \sqsubseteq \forall R.C_i$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C_i^{\mathcal{I}}$
[inverseOf(R_0]	$R = ({}^{-}R_0)$	$R^{\mathcal{I}} = (R_0^{\mathcal{I}})^{-}$
[Symmetric]	$R = ({}^{-}R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^{-}$
[Functional]	$\top \sqsubseteq \leqslant 1\, R$	$R^{\mathcal{I}}$ is functional
[InverseFunctional]	$\top \sqsubseteq \leqslant 1\, R^{-}$	$(R^{\mathcal{I}})^{-}$ is functional
[Transitive])	$Tr(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^{+}$
SubPropertyOf(R_1 R_2)	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
EquivalentProperties(R_1 ...R_n)	$R_1 = \ldots = R_n$	$R_1^{\mathcal{I}} = \ldots = R_n^{\mathcal{I}}$
AnnotationProperty(S)		
Individual(o type(C_1) ...type(C_n)	$o \in C_i$	$o^{\mathcal{I}} \in C_i^{\mathcal{I}}$
value(R_1 o_1)...value(R_n o_n)	$\langle o, o_i \rangle \in R_i$	$\langle o^{\mathcal{I}}, o_i^{\mathcal{I}} \rangle \in R_i^{\mathcal{I}}$
value(U_1 v_1)...value(U_n v_n))	$\langle o, v_i \rangle \in U_i$	$\langle o^{\mathcal{I}}, v_i^{\mathcal{I}} \rangle \in U_i^{\mathcal{I}}$
SameIndividual(o_1 ...o_n)	$o_1 = \ldots = o_n$	$o_i^{\mathcal{I}} = o_j^{\mathcal{I}}$
DifferentIndividuals(o_1 ...o_n)	$o_i \neq o_j, i \neq j$	$o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, i \neq j$

Figure A.1: Axioms and Facts

Abstract Syntax	DL Syntax	Semantics
Descriptions (C)		
A (URI reference)	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
`owl:Thing`	$\top$	$\texttt{owl:Thing}^{\mathcal{I}} = \Delta^{\mathcal{I}}$
`owl:Nothing`	$\bot$	$\texttt{owl:Nothing}^{\mathcal{I}} = \{\}$
`intersectionOf(`C_1 C_2 ...`)`	$C_1 \sqcap C_2$	$(C_1 \sqcap D_1)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap D_2^{\mathcal{I}}$
`unionOf(`C_1 C_2 ...`)`	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
`complementOf(`C`)`	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
`oneOf(`o_1 ...`)`	$\{o_1, \ldots\}$	$\{o_1, \ldots\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots\}$
`restriction(`R `someValuesFrom(`C`))`	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
`restriction(`R `allValuesFrom(`C`))`	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
`restriction(`R `hasValue(`o`))`	$R : o$	$(\forall R.o)^{\mathcal{I}} = \{x \mid \langle x, o^{\mathcal{I}}\rangle \in R^{\mathcal{I}}\}$
`restriction(`R `minCardinality(`n`))`	$\geqslant n\, R$	$(\geqslant n\, R)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x, y\rangle \in R^{\mathcal{I}}\}) \geqslant n\}$
`restriction(`R `minCardinality(`n`))`	$\leqslant n\, R$	$(\geqslant n\, R)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x, y\rangle \in R^{\mathcal{I}}\}) \leqslant n\}$
`restriction(`U `someValuesFrom(`D`))`	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in U^{\mathcal{I}} \text{ and } y \in D^{\mathbf{D}}\}$
`restriction(`U `allValuesFrom(`D`))`	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{x \mid \forall y.\langle x, y\rangle \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$
`restriction(`U `hasValue(`v`))`	$U : v$	$(U : v)^{\mathcal{I}} = \{x \mid \langle x, v^{\mathcal{I}}\rangle \in U^{\mathcal{I}}\}$
`restriction(`U `minCardinality(`n`))`	$\geqslant n\, U$	$(\geqslant n\, U)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x, y\rangle \in U^{\mathcal{I}}\}) \geqslant n\}$
`restriction(`U `maxCardinality(`n`))`	$\leqslant n\, U$	$(\leqslant n\, U)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x, y\rangle \in U^{\mathcal{I}}\}) \leqslant n\}$
Data Ranges (D)		
D (URI reference)	D	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
`oneOf(`v_1 ...`)`	$\{v_1, \ldots\}$	$\{v_1, \ldots\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \ldots\}$
Object Properties (R)		
R (URI reference)	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
	R^-	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
Datatype Properties (U)		
U (URI reference)	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
Individuals (o)		
o (URI reference)	o	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Data Values (v)		
v (RDF literal)	v	$v^{\mathcal{I}} = v^{\mathbf{D}}$

Figure A.2: Descriptions, Data Ranges, Properties, Individuals, and Data Values

Appendix B

The Manchester Pizza Finder

Screenshots of the The Manchester Pizza Finder:

Figure B.1: The configuration screen.

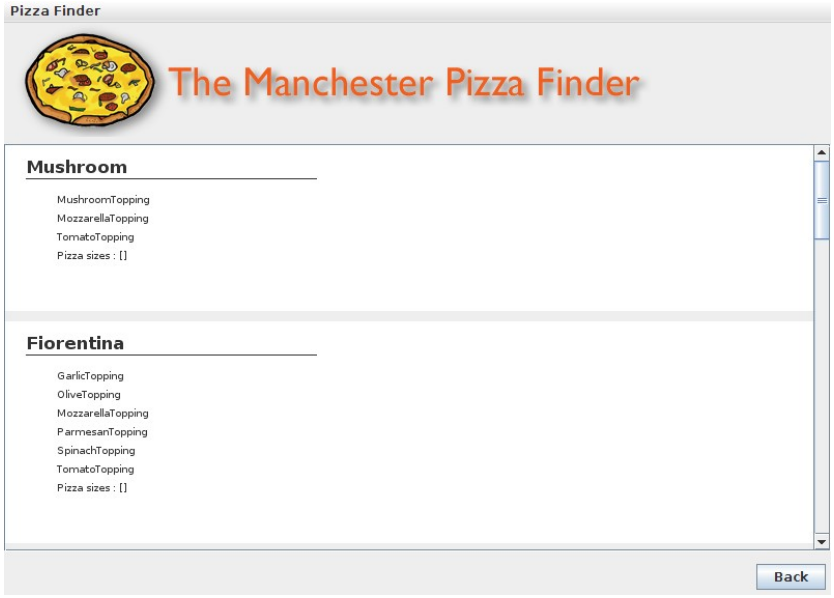


Figure B.2: The result screen.

Appendix C

Design Details of Mooop Prototype

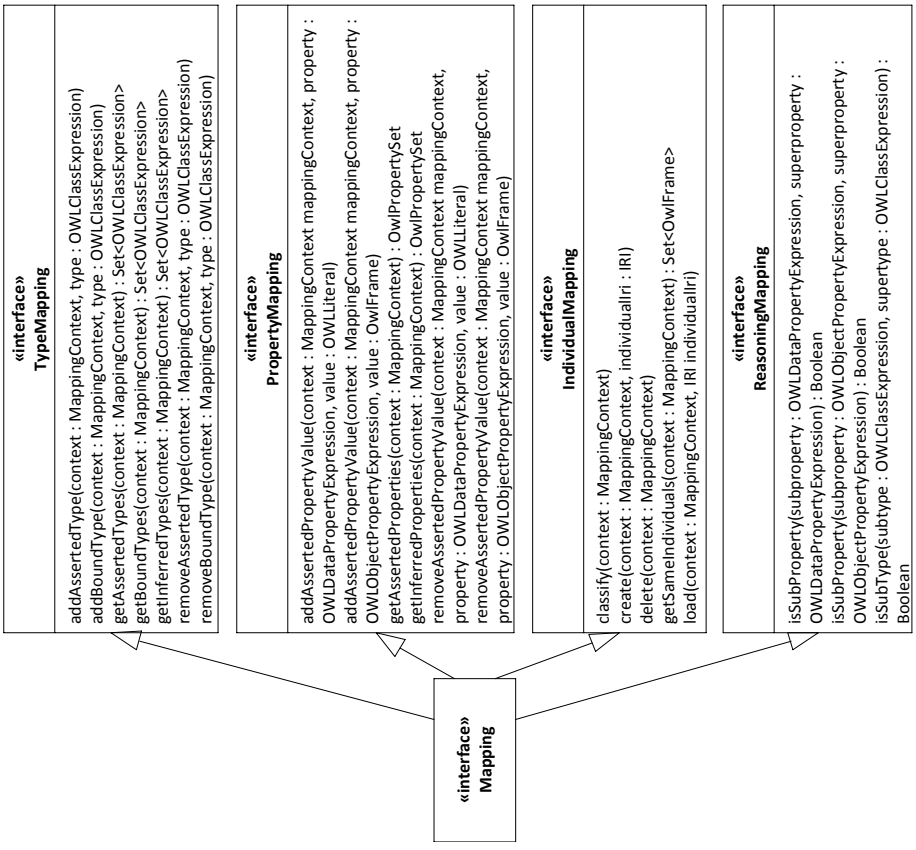


Figure C.1: The mapping interfaces of the Mooop prototype.

```
public class OwlTypeHandler implements MethodHandler {
  ...
  public void addType(final Object type) {
    try {
      final OWLClassExpression classExpression = configuration.
          getDomainFrameConverter()
          .convertOwlClassExpression(type);
      frame.addAssertedType(classExpression);
    } catch (MooopParserException e) {
      throw new MooopRuntimeException(e);
    }
  }
  @Override
  public Object call(final Object obj, final Object[] args, final
      MethodInvocator originalMethod) {
    Object result = null;
    switch (methodType) {
    ...
    case ADD:
      if (args.length != 1) {
        throw new MooopRuntimeException("Parameter not found");
      }
      addType(args[0]);
      // no result
      break;
    ...
    }
    return result;
  }
  @Override
  public void initialize(final Class<?> clazz, final Method method,
      final OwlFrame owlFrame,
      final MooopConfiguration mooopConfiguration) throws
          MooopConfigurationException {
    ...
    frame = owlFrame;
    final OwlType annotation = method.getAnnotation(OwlType.class);
    ...
    methodType = annotation.methodType();
    ...
  }
  ...
}
```

Listing C.1: Example for a method handler taken from OwlTypeHandler.

```
@MooopMethod(methodWrapper = OwlTypeHandler.class)
public @interface OwlType {
  /** type of types which are returned */
  AxiomType axiomType() default AxiomType.ASSERTED_OR_INFERRED;
  /** type of method */
  MethodType methodType() default MethodType.AUTO;
  /** return only most specific types? */
  boolean mostSpecific() default false;
}
```

Listing C.2: The OwlType annotation used for defining access methods to the types of an OwlFrame.

Appendix D

Source Code of the Case Studies

The source code of the implemented prototype of the Mooop concept and the two implemented case studies, pizza configurator and medical patient model, can be found on the attached CD. The code is contained in three Eclipse projects which can be easily imported into an existing Eclipse workspace. The projects are:

Mooop is the prototype of the Mooop framework and contains the following important folders:

- **src** contains the source code.
- **test** contains simple test cases for the framework.
- **lib** contains necessary 3rd party libraries, e.g., the OWL reasoner. We added the three famous reasoners FaCT++, Pellet, and HermiT to the libraries. Therefore, they can be easily used by adjusting the configuration passed to the MooopManager.

Mooop-PizzaApplication is the pizza configurator case study and depends on Mooop.

- **src** contains the source code.
- **test** contains the workflows of the application.
- **eks** contains the ontology for the application.

Mooop-PatientApplication is the medical patient model case study and depends on Mooop.

- **src** contains the source code.
- **test** contains the workflows of the application.
- **eks** contains the ontology for the application.

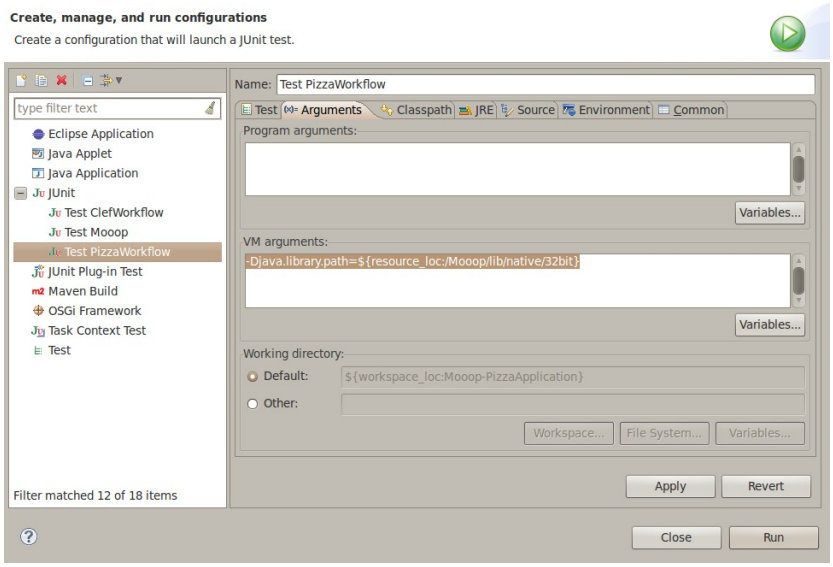


Figure D.1: Run configuration to start the Mooop prototype.

ProfileProject contains the profiling project for the short evaluation of a pizza configurator workflow. After the import of the project the profiling run can be analysed using the "Profiling and Logging" perspective. Please notice that you have to have the *Eclipse Test and Performance Tools Platform*[1] installed.

The two applications are not executable on their own, i.e., they contain no main(...) method. Instead, the workflows can be executed using the JUnit 4 framework which is a part of the Eclipse IDE. However, in order to use the FaCT++ reasoner one has to set a Java parameter so that Java can find the C-library of FaCT++. Therefore, your have to set the following entry as under "VM arguments" in the "Arguments" tab of your run configuration (see Figure D.1):

```
-Djava.library.path=${resource_loc:/Mooop/lib/native/32bit}
```

If you are working on a 64 bit system you have to adjust the argument to point to the directory `64bit` instead of `32bit`. If the Mooop framework and the case studies are not in the same workspace, please ensure that the path for this variable is pointing to the directory containing the files `FaCTPlusPlusJNI.dll`, `libFaCTPlusPlusJNI.jnilib`, and `libFaCTPlusPlusJNI.so`.

[1] `http://www.eclipse.org/tptp/`, accessed 28 Oct 2010